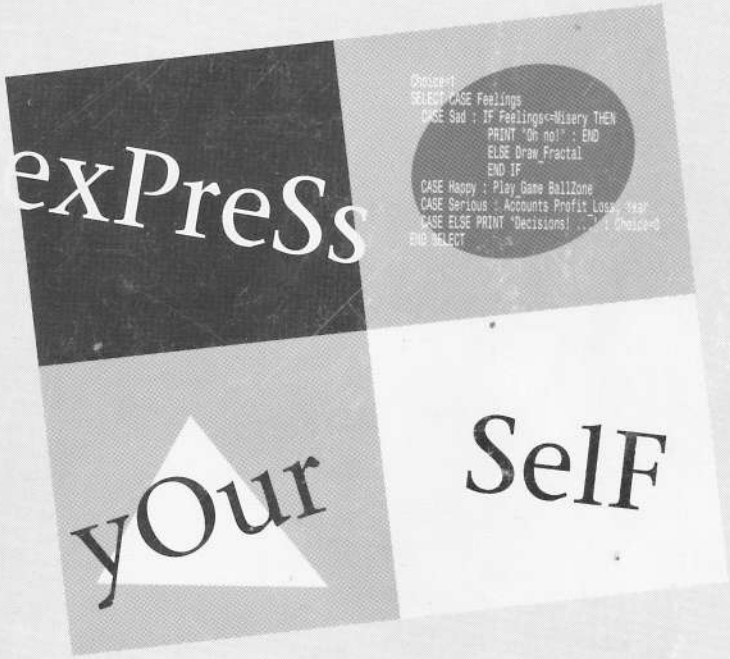
HiSoft BASIC 2 for
ST/STE/TT Computers

ISBN 0 948517 48 4

HiSoft
High Quality Software

# HiSoft BASIC
## version 2




# User Manual

*more software for your Atari ST/STE/TT from*

HiSoft
High Quality Software

# User Manual

## HiSoft BASIC 2 for the Atari ST/STE/TT

***By HiSoft***



***Program:*** designed and programmed by HiSoft.

***Manual:*** written by David Nutkins, Alex Kiernan and Tony Kendle.

HiSoft
High Quality Software

*Published by HiSoft*
*The Old School, Greenfield, Bedford MK45 5DE UK*

First Edition, November 1991 - ISBN 0 948517 48 4

# Table of Contents

# Preface to HiSoft BASIC 2

## Introduction

HiSoft BASIC 2 for the Atari ST/STE/TT range of computers is a powerful and modern version of the BASIC language that gives you a totally integrated and interactive environment for the production of your programs. HiSoft BASIC 2 (from now on, we'll just call it HiSoft BASIC) is a compiled language. This means that your programs are converted into fast running and compact machine code which makes it realistic to use this language for almost any type of application you wish.

HiSoft BASIC is the result of many years design and programming effort and our goal was to produce a BASIC compiler with the following features:

- interactive edit/compile/run cycle, like an interpreter
- compile Microsoft QuickBASIC™ with minimal change
- compile most flavours of BASIC with little modification
- fast compile time and very fast execution time
- full recursive sub-programs and functions with parameters
- many structured statements like `WHILE...END WHILE`, `DO...LOOP UNTIL`, `SELECT...CASE` etc.
- full support for the Atari 680x0 range of computers through the use of libraries, including both low level access and high level graphic and sound functions
- clear error reporting and correction
- no limits on variable size

Thanks to the power and flexibility of the Atari ST/TT and its operating system, we have been able to implement all of these design goals in the HiSoft BASIC compiler and the results are explained in detail in the two manuals that complement the software.

Please spend some time and effort getting to know and learning how to use the manuals so that you can gain the maximum benefit from HiSoft BASIC.

The rest of this section explains how to use the manuals, whether you are a beginner or an expert, how to use your computer to best effect with HiSoft BASIC and, finally, we outline the different type styles that we have used throughout the manuals to (hopefully) make them easy and enjoyable to use.

## How to use the Manuals

This manual is the *User Manual* for HiSoft BASIC; it explains how to work with the package so that you can use it quickly and efficiently. If you are the type of person who prefers to refer to a reference manual when you encounter problems rather than read through descriptions, you may find the companion *Technical Reference* manual more to your taste.

The manuals do not attempt to teach you BASIC, although there is an extended tutorial chapter which starts from first principles and should be helpful to beginners. If you find the tutorial hard work we would encourage you to obtain one of the hundreds of good books on the subject (perhaps through your local library).

We have designed these manuals to tell you about using HiSoft BASIC on the Atari computers. We have packed a great deal of information about the package into the manuals and, in order to help you use them efficiently and easily, we will now plot recommended courses through the manuals for you, whether you are a beginner to BASIC or a seasoned expert.

## A Course for the Beginner

Start by reading the rest of this *Preface*, then *Chapters 1,2, 3* and *5* of this manual in order. You may find that you want to skip some of *Chapter 2* (*The Tutorials*) but we would encourage you to work through the programs to gain experience in using the editor and compiler.

> Passages marked with a sidebar, as this paragraph is, can be omitted at first reading - they provide, mostly, technical or background information.



You should then be ready to begin writing your own programs. You will find it useful to refer to the *Command Reference* (*Chapter 1* in the *Technical Reference* manual) for the commands that HiSoft BASIC allows you to use and *Chapter 3* in this manual for details on using the editor, compiler etc. If you intend to program in GEM, *Technical Reference*: *Chapters 2* and *3* should be consulted.

At some stage we recommend reading the *Concepts* chapter to gain an overview of the language features of HiSoft BASIC.

The Appendices are mainly for reference and you will only need to dip into them occasionally although we would recommend reading *Appendix B* in this manual (*Hints and Tips*) at an early stage.

We hope you find HiSoft BASIC easy and friendly to use, please do not hesitate to write to us with any suggestions for improvements and/or alterations.

## A Course for Seasoned BASIC Programmers

If you are already familiar with BASIC programming and also with the Atari 680x0 computers then we recommend that you use the manuals in the following way.

Firstly, finish reading this *Preface* and then move on and read *Chapter 1* and *Chapter 3*. If you are eager to start programming then simply double-click on HBASIC2.PRG (on your working disc!) and refer to *User Manual : Chapter 3* and *Technical Reference : Chapters 1, 2, 3, 4* and *5* as and when necessary. It is useful to read the *Concepts* chapter in this manual at some stage to appreciate the language features of HiSoft BASIC.

We would encourage you to work through at least the latter part of *User Manual : Chapter 2* (*The Tutorials*) especially the section on the use of the *HiSoft GEM Toolbox*.

If you intend programming with GEM then you will need to consult *Technical Reference : Chapters 2* and *3*. When you are more confident with GEM you may like to look at *Technical Reference : Appendix G*, detailing how to write desk accessories. For information on linking with C and assembler language programs, look at *Technical Reference : Appendix F*.

The other Appendices are for general reference and it is worth glancing through all of them to acquaint yourself with their contents.

Good luck, we hope you find HiSoft BASIC a powerful, flexible and easy-to-use development system. Of course, we welcome any written comments you may have on how we might improve both the program and the manuals.

## System Requirements

HiSoft BASIC will run on any Atari 680x0 computer (ST/STE/TT) with at least 512Kb of memory and a double-sided disk drive. The programs should work in any resolution on all commercially available monitors.

Users with only 512Kb of RAM may run out of memory when attempting to compile larger programs or in other circumstances. To minimise the former problem the compiler must be given as much memory as possible to run in. The installation of even a small RAM-disk on a 512Kb machine will restrict HiSoft BASIC, and if you use any desk accessories you should consider removing some of them unless you really do need them.

You will need 1Mb of memory to work with the *HiSoft GEM Toolbox* and to run any other resident program, such as the resource construction set, in the interactive environment.

If you are short of memory, remember that the least memory hungry thing is to compile a one line program (consisting of a REM $include statement) from the desktop.

Upgrades to a megabyte of memory are available at very reasonable prices and we strongly recommend this, not just for HiSoft BASIC but for general use too.

## Typography

In order to make the manuals easy to read and to convey the maximum information as clearly as possible, we have adopted certain typefaces and typestyles throughout the manuals. These are used as follows:



## Typefaces

| | |
|---|---|
| Palatino | General text. |
| *Futura oblique* | Chapter and sub-Chapter headings and references to them. |
| Monospace | Used to show something that is typed in at the keyboard or displayed on the screen. Predominantly used in program listings and references to function names, variables etc. |
| Avant Garde | Used for filenames, menu selections and button names. Also used to denote legends on single keys such as Alt (the Alternate key) and Ctrl (Control). |

## Typestyles

*Italic* Used for emphasis. and in syntax descriptions to show something to be filled in e.g. `INT (numeric_expression)` where *`numeric_expression`* is to be replaced by an expression when `INT` is used in a program.

## Special Characters

| | |
|---|---|
| [] | Within syntax descriptions, information enclosed in [] is optional e.g. `[CALL] sub_program_name` means that you can call a sub-program `TEST` by `CALL TEST` or simply by `TEST`. |
| {} | Within syntax descriptions this indicates a choice of one or more options, each separated by a vertical bar (\|) e.g. `PRINT [expression_1] [{;\| \|,} expression2]...` indicating that expressions in `PRINT` statements may be separated by a semi-colon, a space or a comma. |
| ... | Indicates repetition in syntax descriptions. |

. Vertically-spaced dots show that some part of a
. program has been omitted.
.

## Acknowledgements

The trademarks (both registered and otherwise) of various companies are used throughout its manuals. In particular:

*QuickBASIC* is a trademark of Microsoft Corp.

*TurboBASIC* is a trademark of Borland International Inc.

*HiSoft BASIC, Power BASIC, FirST BASIC, DevpacST, GenST, MonST* and *WERCS* are trademarks of HiSoft.

*GEM* is a registered trademark of Digital Research Inc.

*ST, STE, TT, TOS* are registered trademarks of Atari Corp.

We acknowledge any other trademark used but not listed above.

We would like to thank the following people for their invaluable help in the production of HiSoft BASIC and the manuals:

Stephan (42) Somogyi for his wit and unfailing ability to keep us smiling, Julia for holding the fort when lesser people would have deserted, the lady with the dog (who was that girl?) and all the staff at The Old Bell!



# Chapter 1 Introduction

## Always make a back-up

Before using HiSoft BASIC you should make back-up copies of the distribution disks and put the originals away in a secure place; safe from extremes of temperature, magnetic fields, moisture and children! The disks can be backed-up using the Desktop or any back-up utility - before making any backup always write-protect the master to prevent accidental erasure.

The disks are not copy-protected to allow easy back-up and to avoid inconvenience; remember though that the software and this manual are protected by international copyright laws and you are only permitted to copy the software for your own personal use. If this sounds officious, look at it another way - if you give away copies of HiSoft BASIC to your friends we will not receive enough revenue from the sale of the package to improve this and other products. We want to help you, please help us in return.

## What's on the Disks

HiSoft BASIC is supplied on three double-sided disks.

Disk 1 contains all you need to start writing and running programs whilst disk 2 contains sundry tools including a command line version of the compiler, the profiler etc. Finally, Disk 3 contains TT versions of the compiler and libraries plus last minute additional files. You should note that there may be extra files on any of the disks - please consult the README.TXT file on Disk 1 for details.

There are a large number of programs and other files on these 3 disks but you will not need all of them - in fact many programmers will only ever use Disk 1.

## Disk 1

| | |
|---|---|
| HBASIC.PRG | The integrated version of the compiler |
| HBASIC.LIB | The library file that is loaded by the integrated compiler |
| HISOFTED.INF | The editor and compiler settings file |
| WERCSBAS.PRG | The resource construction set |
| WERCSBAS.RSC | The resource file for WERCSBAS |
| WERCS.INF | Settings file for WERCSBAS |
| HB2INST.PRG | A quick installation program for HiSoft BASIC |
| DESKTOP.INF | Desktop settings file so that HiSoft BASIC will be automatically loaded when the disk is booted on machines with TOS 1.04 and above |
| README.TXT | The latest changes to the package |

In the AUTO folder:

| | |
|---|---|
| HRAMDISK.PRG | A small RAM disk for 1Mb users; delete this from your AUTO folder if you have only 512Kb of memory |

The source code and pre-tokenised files of the *HiSoft GEM Toolbox* can be found in the HGT folder (you can delete this from your working disk if you have only 512Kb of RAM):

| | |
|---|---|
| HGT.T | Pre-tokenised version of the full GEM toolbox. Some other .T files will be on this disk & some more may be in the HGTTUTOR folder on disk 2 |
| HGT.BAS | A file that 'includes' all of the GEM toolbox routines |
| TOOLBOX.BAS | The core routines of the toolbox |
| DIALOG.BAS | Routines used in dialog boxes |



| | |
|---|---|
| EXEC.BAS | Code for running other programs |
| FILESEL.BAS | Code for use with file selectors |
| IMAGE.BAS | For use when handling images |
| IMAGWIND.BAS | Provides support for images with windows |
| MENU.BAS | Routines for use with menus |
| NEWDESK.BAS | Code for customising the Desktop |
| OBJWIND.BAS | Needed when coding object windows |
| TEXTWIND.BAS | Text window routines |
| WINDOW.BAS | General window handling code |
| ACCTBOX.BAS | Desk accessory version of TOOLBOX.BAS |
| GEMAES.BH | The AES constant header file, needed by TOOLBOX.BAS |

In the TUTORIAL folder there are the source of many of the programs used in the *Tutorials* chapter, including the following:

| | |
|---|---|
| ABC.BAS | The GEM example worked through in the *Tutorials* chapter |
| HANOI.BAS | The famous rings of Hanoi puzzle |

## Disk 2

In the TOOLS folder:

| | |
|---|---|
| HBASIC.TTP | The command line version of the compiler (without the editor) |
| MONSTB.PRG | The ST version of the medium level debugger |
| BUILDLIB.TTP | Library file builder for customising HBASIC.LIB |
| CHECKST.PRG | Diagnostic program for displaying the ROM version no etc. |
| CTOBAS.TTP | Simple C header file to BASIC header converter |
| WCONVERT.TTP | Program to convert a name file from other RCSs to WERCS' .HRD format |
| WCONVERT.PRG | GEM version of WCONVERT.TTP |
| WIMAGE.PRG | Program for importing Degas and Neochrome images into WERCS |
| PROFILE.TTP | HiSoft BASIC 2 profiler |
| RAMINST.PRG | The installation program for the HiSoft RAM disk |
| RAMINST.RSC | The resource file for the HiSoft RAM disk |

In the LIBS folder:

| | |
|---|---|
| GEMAES.BH | AES constant header file |
| BIOS.BIN | BIOS library |
| GEMAES.BIN | GEM AES library |
| MENU.BIN | The Menu library; now contains the Menu function |
| GEMDOS.BIN | GEM DOS library |
| GEMVDI.BIN | GEM VDI library |
| HBASLIB.BIN | Core BASIC runtime library |



| | |
|---|---|
| LIBDEMO.S | Example source of a library |
| LIBRARY.H | Header file for writing your own libraries |
| STESOUND.BIN | STE Sound library, not included by default |
| XBIOS.BIN | Extended BIOS library |

In the COMPAT folder:

This folder contains several small include files for your use if you are converting source files previously used with other versions of BASIC.

The *HiSoft GEM Toolbox* examples mentioned in the *Tutorials* are in the HGTTUTOR folder.

In the EXAMPLES folder there are further examples of the use of HiSoft BASIC, which you should try out for yourself.

The HGTEXAM folder within the EXAMPLES folder includes many exanples of the use of the GEM Toolbox, including:

| | |
|---|---|
| ADDRESS.BAS | Address book example program |
| ADDRESS.RSC | Resource file for the address book |
| ADDRESS.HRD | Resource name file for the address book |

You will need the files in the LINK folder only if you are interested in linking HiSoft BASIC with assembler or C. It contains the following files:

| | |
|---|---|
| HBCUTIL.O | C Utility functions |
| FREQ.BAS | Example program without linking |
| FREQLINK.BAS | Example program with linking |
| GETCH.S | Assembly language part to link with FREQLINK.BAS |

## Disk 3

In the TT folder:

| | |
|---|---|
| HBASICTT.PRG | The integrated version of the TT compiler |
| HBASICTT.LIB | The standard library for the TT version of the compiler |
| HBASICTT.TTP | Standalone version of the TT version of the compiler |
| MONTTB.PRG | TT version of the debugger |
| LIBS*.* | TT versions of the libraries corresponding to the ST versions on disk 2 |

There may be other files.

The minimum requirement for your work disks is the first three files listed on disk 1 - they *must* always be present and *always* in the same folder. The presence of all the other files is optional.

To run HiSoft BASIC, double click on the HBASIC.PRG icon from the Desktop, or run it from a shell. When it has loaded a menu bar will appear and an empty window will open, ready for you to enter and compile your programs.

## Making a Working Disk

HiSoft BASIC is supplied on three double-sided diskettes but you will not always need to use all the disks; which programs and files you will require will depend on your programming application, the amount of memory that you have and other factors. It is a good idea to make a working disk (or folder on a hard disk) in which you place the programs and files that are suited to your particular computer set up.

To help you do this, we have supplied an installation program on Disk 1 called HB2INST.PRG; double-click on this file (from your back-up of disk 1) for installation details.



## Registration Card

Enclosed with this manual is a registration card which you should fill in and return to us in order to register your purchase of HiSoft BASIC. This will entitle you to a free period of technical support and will enable us to keep you informed of future developments to our software.

For details of our technical support services, please refer to *Appendix C* in this manual.

You will need to quote your serial number (to be found on the disk label on disk 1) to obtain technical support and you may find it useful to make a note of it here:

| Serial No. |
|---|

## The README File

As with all HiSoft products HiSoft BASIC is continually being improved and the latest details that cannot be included in the manual may be found in the README.TXT file on Disk 1

This file should be read at this point, by double-clicking on its icon from the Desktop and then clicking on the Show button. You can direct it to a printer by clicking on the Print button. Alternatively, you can read it by loading it in to your favourite editor.

# Chapter 2
# Tutorials

BASIC is just one of many computer languages that you can buy for your Atari ST/STE/TT computer (you could contact HiSoft for up to date details of the range available) but BASIC is deservedly popular for its ease of use and the speed with which a beginner can learn to produce results.

On the other hand there are two common criticisms of BASIC. Firstly, many people feel that its very flexibility and tolerance encourages poor programming habits compared to highly structured languages such as Pascal. Secondly, BASIC is often the slowest language you can use, which makes it unsuitable for many time-dependent applications such as games and graphics.

Our new BASIC, HiSoft BASIC 2, goes a long way towards solving both of these problems. We have taken the best qualities from Microsoft BASIC, Pascal and other languages and combined them together to produce an implementation that is capable of running traditional BASIC programs but can also be used to produce a source that is highly structured, easy to follow, and simpler to debug. It is also extremely powerful, giving you the ability to control almost every feature of your computer, hardware and software ... and, above all, it is fast.

Many people familiar with older versions of the language will have some re-learning to do. Bad habits and unnecessary features, such as line numbers, can now be forgotten, although you can use line numbers if you really want to. HiSoft BASIC does not mind if you want to take things gently and continue using them for a while.

To help you get to grips with BASIC and, in particular, HiSoft BASIC, we have written a two-part tutorial which forms the rest of this chapter. The first part is aimed at users who have not used BASIC very much before and starts from first principles; experienced users may find it useful as a refresher course.

Following this there are worked examples of using the *HiSoft GEM Toolbox* supplied with HiSoft BASIC; everybody should work through this since, once mastered, it makes GEM programming far easier than ever before.

> Throughout the tutorials you will see some text marked with sidebars, as this paragraph is marked. These passages are diversions which need not be read the first time through but contain supplementary information which you should find useful once you have become more confident with HiSoft BASIC.

# BASIC Tutorial

The examples used in the following tutorial are designed to illustrate the power of HiSoft BASIC. They will therefore tend to use its structured elements as much as possible, but do remember that you can also type in many programs from books and magazines that would work with more inferior versions of BASIC. They should run, and run much faster, with hardly any alteration at all; if there are problems, HiSoft BASIC will point you towards which lines are causing the trouble, quickly and simply.

All computer programs are simply tools which allow us to control the actions and reactions of the computer itself. HiSoft BASIC has been written to be as compatible as possible with several different versions of the language - notably the ST BASIC that came supplied with older Atari STs and also Microsoft QuickBASIC which is one of the most highly respected implementations of the language available for the IBM PC range of computers (and compatible machines). Language elements have also been added from Borland's Turbo Basic for the PC and AmigaBASIC.

We aimed at this compatibility for two reasons. Firstly, anyone who has already learnt to program in one of these different versions of BASIC will be quickly able to work with the HiSoft BASIC language. Secondly, programs that are have already been written using these versions will run with little or no alteration under HiSoft BASIC (but of course they should run much faster!).



The latter feature is particularly important to anyone who intends to run a specific program, one that was originally intended for the IBM PC for example. It will also allow anyone who wishes to learn more about programming to buy tutorial books that were aimed to be used with one of these versions of the language.

However, for anyone who is new to programming it is not all good news. HiSoft BASIC has had to conform to the language standards used by at least two different systems, and these in turn have built upon previous versions of BASIC and so on. As a result not all commands are as self explanatory or as easy to use as we, or you, might wish. One area of difficulty is that, because HiSoft BASIC is compatible with so many other BASICs, it has a large number of Reserved Words that you should avoid using as labels or variables e.g.

```
Loop:
PRINT "Welcome to HiSoft BASIC!"
GOTO Loop
```

If you typed this program in and tried to execute it using HiSoft BASIC, you would find that an error will occur on line 3. This is because `Loop` is a reserved word in HiSoft BASIC (you will learn how to use it later) and thus you cannot use it as a label or variable. You must watch out for this type of error since you can easily confuse HiSoft BASIC by using its reserved words incorrectly. See *Technical Reference: Appendix A* for a list of the Reserved Words.

> Experienced programmers, who are sure that they know what they are doing, can remove a Reserved Word so that it can be re-defined. You do this with a line like:
>
> `REM $option ~ PRINT`
>
> to remove `PRINT` from the Reserved Word list. You can then create a new sub-program or function called `PRINT`.

This tutorial is intended to be a complement to the *Command Reference* section of the *Technical Reference* manual. Unlike the latter, here keywords are grouped logically by their general function rather than alphabetically. We have tried not to duplicate information that is perfectly obvious from the *Command Reference*, but rather to summarise the way that certain keywords are related to each other and interact with each other. At times it has been inevitable to make forward references to keywords or programming techniques that have not yet been covered in detail, but we have tried to keep these instances to a minimum.

Please consult the *Command Reference* chapter in the *Technical Reference* manual frequently as you work through this tutorial - you will find understanding will dawn more quickly and more easily if you supplement your knowledge in this way.

After the first few programs presented in this tutorial it will be assumed that you know how to use the HiSoft BASIC editor; the various demonstration programs will be presented without constant reminders of how to enter and run them although we will give examples of editing techniques etc. from time to time. If you need further help, you should consult *Chapter 3* on using the editor.

## The Building Blocks of BASIC

We can look on each component of the BASIC language as a building block from which we can, step by step, construct an entire program. As we shall see later, the structured aspects of HiSoft BASIC make it particularly easy to develop very long and complex tasks as a collection of small modules, each one of which performs a small part of the whole and which can be tested and perfected in isolation.

The smallest component of the BASIC programming language is known as a *keyword*. Each of the keywords does a certain job, although some of them must first be given some information to work on, and many of them only function when used in conjunction with certain others.

For your reference, a list of all keywords used by HiSoft BASIC is given in *Technical Reference: Appendix A*. You must ensure that you do not try to use any of these words to represent variable or subroutine names or HiSoft BASIC will become confused as to which you really mean.



The function and syntax of every single keyword will be detailed in *Technical Reference: Command Reference*. This tutorial section will not attempt to cover every one of these again, but rather try to illustrate the way that some of these building blocks can be joined together into useful routines.

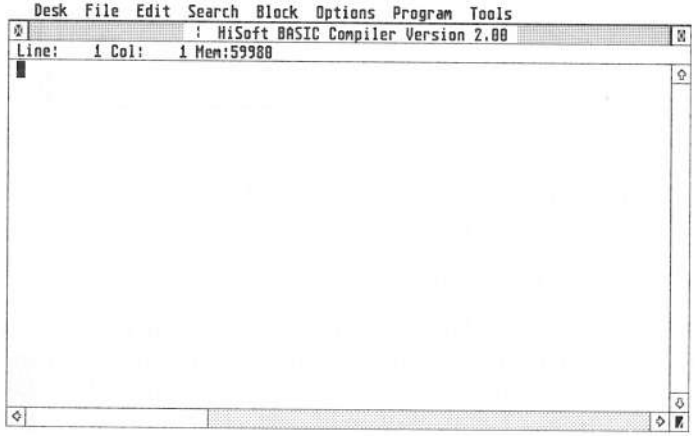
The most important thing is that you learn to 'think like a programmer'. Once you begin to see how small programs work, how the individual commands fit together to produce the desired end result, you will quickly be able to pick up new keywords as and when you need them.

## Some Simple Commands

There are many very simple tasks you can ask your computer to do which really form the 'bread and butter' of computer programs. The simplest example to start with is unquestionably a command like the BEEP command which, no prizes here, causes the computer to emit a short note. We'll try it now and enter a short program.

Start by locating the HBASIC2.PRG file on your working disk or hard disk (see *Chapter 1*), then double-click with the mouse on the HBASIC2.PRG icon to run HiSoft BASIC.

After a short while, the editor window will appear:

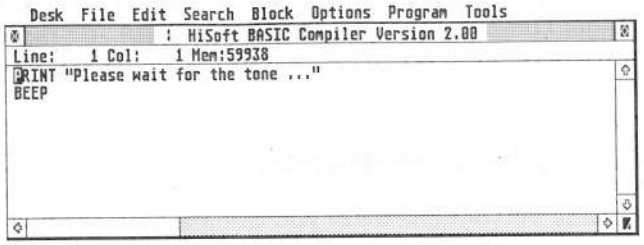



*HiSoft BASIC loaded*

Now type the following, pressing the Return key at the end of each line:

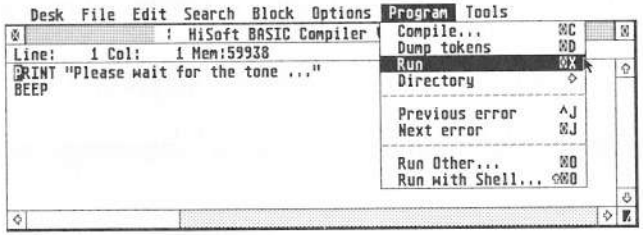
```
print "Please wait for the tone ..."
beep
```

The screen should look like this:




*Your first HiSoft BASIC program*

Now move the mouse up to the Program menu, press the left button and, holding the button down, move the mouse until you have selected the Run command. Now release the button.




*Running your first program*

This will run your program which should clear the screen, display its message and beep ... impressed?

Ok, but it *was* simple to do.

Press a key and select New from the File menu to create a new window - note that this is window number 3. You may be wondering what window 2 is - press Alt-2 (this means hold the Alternate key down and press 2) and you will see. HiSoft BASIC puts its messages in a separate window when it is working so that you can inspect them afterwards. Ok, now press Alt-3 to go to our new window.

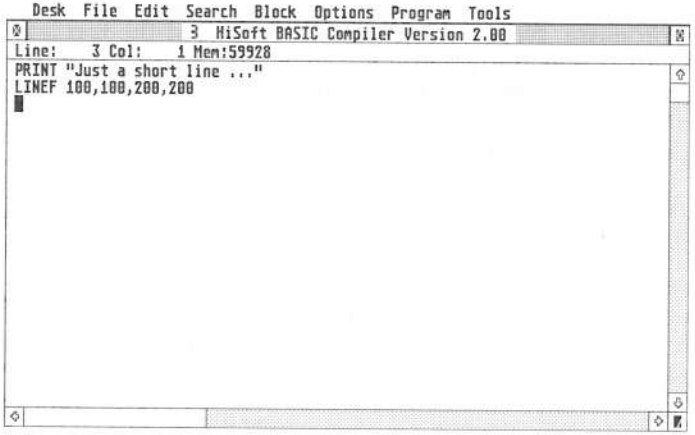


We'll move straight on to a more complex (and Atari specific) command,; LINEF draws a line on the screen. By looking in the *Technical Reference : Command Reference* chapter you will see that this keyword can be given several parameters which control the start location and finish location of the line. We will be dealing with LINEF and other graphics commands again later in the tutorial, so do not try and take it all in at this stage. Just try this simple command for comparison with BEEP. Remember that it does not matter whether you type the HiSoft BASIC commands in upper or lower case or even a mixture of the two; LINEF, linef and lineF are all the same to HiSoft BASIC.

> The editor will upper-case HiSoft BASIC Reserved Words automatically if you have selected Show BASIC tokens in Preferences... under the Options menu. This is useful as a quick syntax check after you have completed each line.

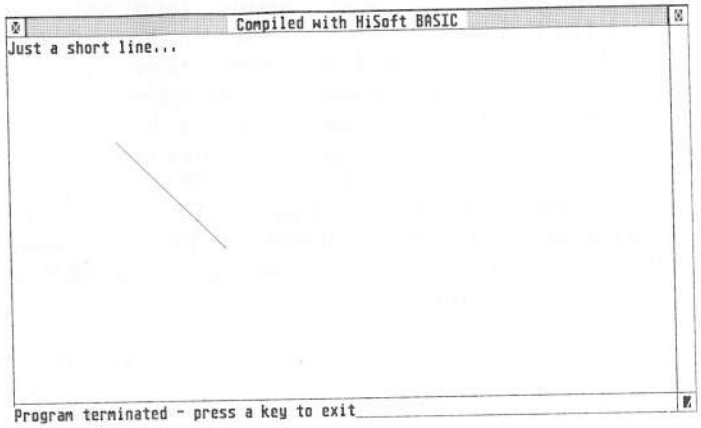
```
PRINT "Just a short line ..."
LINEF 100,100,200,200
```

Type this in:




*Your second program, using LINEF*

You can run this as before, by selecting Run from the Program menu or you can run it by using a *keyboard shortcut*, Alt-X - it achieves exactly the same as selecting Run but some people prefer the keyboard to the mouse. There are many keyboard shortcuts within the HiSoft BASIC editor.




*The LINEF program running*

Notice how HiSoft BASIC automatically gave you a window when you ran a program that used some of GEM - this is very useful but you may wish it not to happen, we will cover this later.

Press a key so that you are back looking at the source code of the LINEF program. Now click in the *Close* box at the top left of the window to forget the program - click OK when the dialog box appears. Do this with window 2 (the messages window which is read-only so that no warning box will appear) and window 1 (the BEEP program) so that you have closed all the windows.

Press Alt-1 to open a new, empty window ready for another program.

The commands that you can give the computer must be presented in a fairly rigidly defined style and format known as the *command syntax*. The correct syntax for every command you use is given in the *Command Reference* chapter which can be found in the *Technical Reference* manual. Program syntax can be compared to the grammar of the English (or any other) language; if you want to make yourself understood to other people you should speak and write reasonably correct English. In the same way, you must be careful to write your BASIC programs correctly for HiSoft BASIC to understand what you mean.



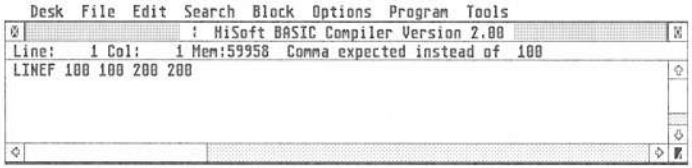
One thing computers are very poor at (and people are normally rather good at) is guessing what we mean when we present them with ambiguous or incomplete instructions (which is why a computer can do many engineering tasks but will probably never have a career in politics).

For example we cannot use the commands:

```
LINEF 100 100 200 200
LINEF 100:100:200:200
LINEF (100)(100)(200)(200)
```

and expect the computer to do what we wanted, we must obey its own rules and not invent our own. Try typing in the first line above and running it (Alt-X, remember?).

You should find that an error message appears and you are asked to `Continue(Y/N)?` Press N or n and you will be returned to your source code window:




*The editor reporting an error*

Something has gone wrong, the program did not run. Look to the top right of the window, you should see an error message (`Comma expected instead of 100`) - was that what you expected? HiSoft BASIC knows what it wants and has told you quite clearly - unfortunately, error messages are not always so obvious. Close the window by clicking in the *Close* box or by pressing Ctrl-W or by selecting Close on the File menu - these all do the same thing.

Errors in syntax rather than logic are among the most common mistakes beginners to computing make. Where it can, HiSoft BASIC will make every attempt to catch these errors and explain to you what it cannot understand (if you think about it, to explain one's own misunderstandings is something of a paradox so please be tolerant if HiSoft BASIC does not always highlight the exact problem).

At this stage it is useful to cover a valuable keyword; the REM or 'do nothing' command. REM is actually short for REMark and anything you type on a program line after this keyword will be ignored by HiSoft BASIC, whether it contains valid keywords or just plain ordinary English. The main use of the REM command is to allow you to add notes to your program which explain what it is the next section of the program is intended to do.

REM is also invaluable for catching obscure mistakes (or 'bugs') in your program where you have got the syntax of the program correct but there is an error in the logic which the compiler may not be able to spot. If you have commented your program clearly it is often possible to quickly isolate and identify where the problem lies.

> REMs can also be used to 'switch off' sections of the program that you have tested and know work correctly, so that the parts still under development can be run in isolation.

There is an abbreviation for REM - the ' (single quote/apostrophe) mark. This is particularly useful for commenting out program lines, because it is quicker to insert.

Much more useful than our friends BEEP and LINEF, but also more complex, is the PRINT command which is used to get the computer to put information on the screen, as we have already seen above. Extremely impressive tabular effects are possible using the most sophisticated options of this command but, since it is also an instruction that you will be using almost every time you program in BASIC, we might as well learn it quickly - it is after all one of the few means by which we can get the computer to tell us the results of any calculations or tests it has performed.

In its simplest form the PRINT command needs no parameters beyond some indication of exactly what it is that you want displayed. One way to try it out is to use the computer as a, rather expensive, pocket calculator. Try entering some of the following short commands. Remember, use Alt-X to run the program and Close (on the File menu) or Ctrl-W to clear out the text.

For deleting short programs, alternatives to closing the window are: position the text cursor at the end of the line and use the Backspace key repeatedly or position the cursor within the line and use the delete line command, which is Ctrl-Y.



```
PRINT 23+45

PRINT 12*6

PRINT 99/9
```

Of course it is possible to print text as well as numbers as so:

```
PRINT "Ten times twenty six equals"
PRINT 10*26
```

To signal to BASIC that you want it to print the text as it is given, rather than attempting to look the words up as potential numeric *variables* (see below) the text must be enclosed in double quotes (" "). Again this is an example of the need to use the correct syntax to avoid ambiguity.

## Layout of Programs

Before we get too involved it is worth pointing out some simple points about the layout of BASIC programs.

The highly advanced structured programming features provided by HiSoft BASIC are intended primarily to allow the programmer to work in a clearly defined and modular way. These make it very much easier to follow the logic of what is trying to be achieved. However it is not only the language keywords that make this possible; the way that the listing is organised, on screen or on paper, is very important so that the programs are easy to follow.

HiSoft BASIC does not care whether the keywords you type are in upper or lower case, but they will be much easier for you to spot and to follow if upper case is used; in fact the editor helps you by upper-casing keywords automatically if you check Show BASIC tokens in the Preferences box on the Options menu. The editor also does not mind how many extra spaces are used to indent different lines of the listing - make full use of this feature and set out your modules so that the eye can easily follow the hierarchy of the program. Again the editor can help by automatically indenting lines; this is described in more detail later on.

Like all modern versions of BASIC it is possible to put more than one command on each line, as long as each command is separated from the next by the colon character (:); this can be very useful to maintain the logic of the program.

Unlike many versions of the language, HiSoft BASIC does not *need* line numbers; you can use them if you wish or avoid them like the plague!

## Using Simple Variables

The power of a BASIC calculator program such as we have written above can be greatly increased by the use of simple numeric variables. Again there is much more to the subject of variables than we will cover here, but in their simplest form they are just letters or short names that can be assigned a value, and used to represent that value in calculations (just as you would do in algebra, remember that?).

Any text string or number that is used explicitly rather than assigned to a variable is a constant.

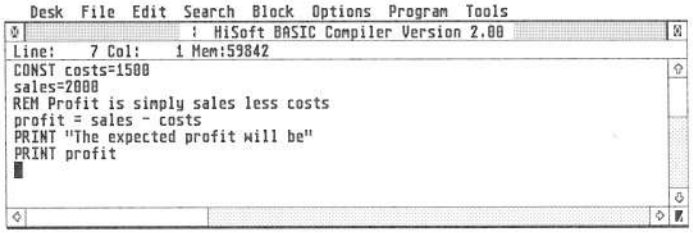
As well as being useful and flexible tools for use in calculations, variables have the useful facility that they can be given descriptive names such as `profit` and `sales` which make it very much easier to follow the logic of the computer listing to see what each line is doing.

So valuable is this feature that HiSoft BASIC also allows constants to be given descriptive names as well. To denote that a given name is to be treated as a constant and not a variable we must use the keyword `CONST`.

Try this example:

```
CONST costs = 1500
sales = 2000
REM Profit is simply sales less costs
profit = sales-costs
PRINT "The expected profit will be"
PRINT profit
```






*Using variables*

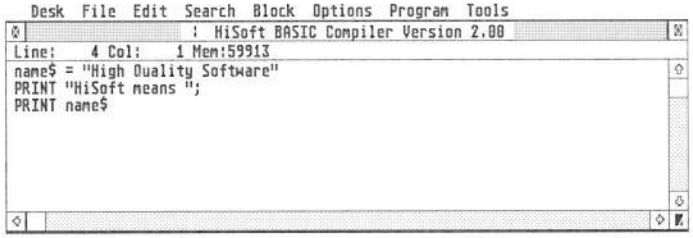
As a simple illustration of the usefulness of variables edit the first two lines of the program to give different values for `costs` and `sales`; do this by placing the cursor at the end of the first line, pressing Backspace to delete the `1500` and entering another value. Repeat this for `sales`. Run the program and then close the window.

Later in this tutorial we will see that the program can be extended to automatically ask you for values for these variables as the program is run.

Variables can also be defined that hold text rather than a number. To warn BASIC that this is what you intend to do, syntax demands that all text variable names must end in the $ character. In computer speak '$' is often pronounced 'string' and represents 'a string of letters joined together'. The text that you wish to enter into the variable must again be enclosed in quotes.

An example is:

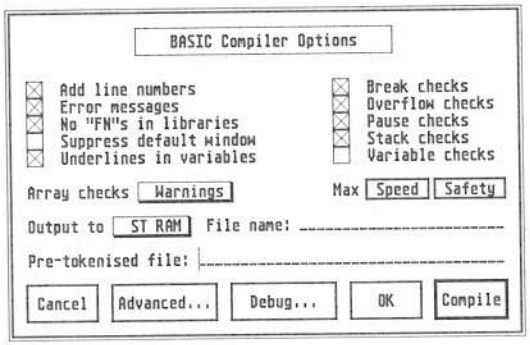
```
name$ = "High Quality Software"
PRINT "HiSoft means ";
PRINT name$
```




*A quality program ...*

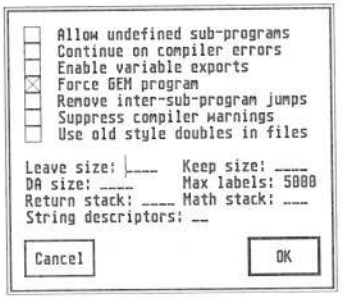
Before we run this program we'll take a slight diversion. You may have noticed when you ran the previous program that there was no window - the text was printed at the top of a blank screen. This is because the program didn't use any GEM commands and therefore a window was not appropriate. However, if you wish to force a GEM window to appear in a program that doesn't use GEM you can, like this.

Select Compile... on the Program menu and the following box will appear:




*The Compile... dialog box*

Now click on Advanced...

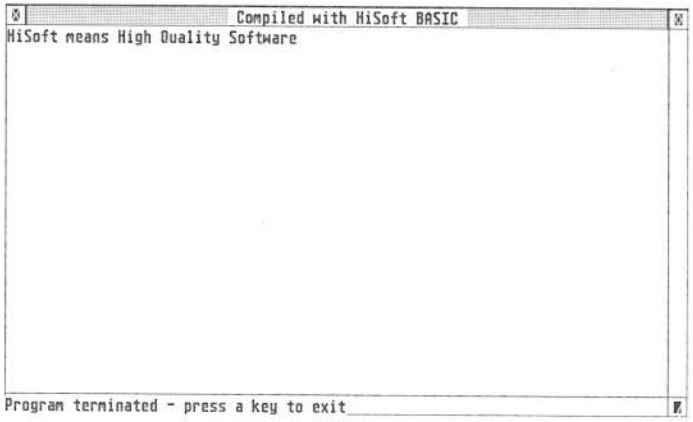



*Forcing a GEM program*

Finally, click in the line Force GEM program until the box next to it has a cross in it (this is called *checked*). Click OK and OK in the Compile... box.



Now run your program.




*A non-GEM program, in a window*

You may have noticed something else different about the way that the above program produced its output i.e. it is all contained on one line. This is the result of the semi-colon (';') at the end of the second line.

The semi-colon is one of many possible ways we can control the way that `PRINT` statements are displayed, and these will be detailed later. For the time being though it is useful to know that the semi-colon ensures that any following print statement is added onto the end of the existing one, rather than starting a completely new line. We will be using this feature a lot when we start to look at programs that would otherwise very quickly fill the screen with information. Notice also the trailing space at the end of the second line - we need this to avoid running the words `means` and `High` together.

The above programs are useful illustrations of how variables work, but they do not really reflect the way that we use them in real programs. The truth is that we often do not know in advance what the value of a variable is to be before the program is run. It is possible, as above, to edit the list of variables and recompile the program every time we use it, but a much more powerful and flexible method is of course to get the computer to ask you for the variable's current value whilst the program is running. This is done using the `INPUT` keyword.

If you use the keyword INPUT followed by a variable name, the computer will display a question mark on the screen and pause the program until it has had an answer from you, as in this example:

```
INPUT x
y = x * 10
PRINT "Ten times"; x ; "equals"; y
```

Of course, in a long and complex program we will want to be much more friendly than that - question marks appearing at random to request unspecified information can be rather confusing!

The INPUT statement therefore allows you to enter a explanatory line of text that will be displayed together with the question mark.

```
INPUT "What is this year's profit "; x
```

There are of course very many other simple BASIC commands that each achieve a certain effect. There really is no point in detailing each of them here, but look through the *Command Reference* section in the *Technical Reference* manual and find some more to try. Unless you are very unlucky with your choice, absolutely no harm can be done, but it is just as well to avoid using the commands BLOAD, PEEK and POKE for the time being.

Here is an example of how to read the syntax of each command so that you know what HiSoft BASIC expects.

## Reading the Syntax

```
INPUT [;]["prompt"{;|,}] variable_list
```

First of all, INPUT is the actual command and *must* be present to achieve anything at all.

```
[;]
```

The square brackets ([ ]) around the semi-colon indicate that its use is optional, you can leave it out and the syntax will still be valid. If you include the semi-colon then the cursor will stay on the same line after the INPUT statement is finished i.e. after the user has pressed Return.

```
["prompt"{;|,}]
```



Again, the square brackets denote that the whole entry is optional. If you choose to use it though, you must enclose a message in double quotes (" ") and finish the message with either a semi-colon or a comma. The curly brackets ({}) denote a choice with the vertical bar (|) separating the choices that are available in this case. In fact, using the semi-colon to end the prompt string adds a question mark to the end of the string, while using the comma will not add anything to the string.

```
variable_list
```

Finally, the INPUT statement must be finished with a list of variable names to which you want to assign data - this is not optional.

With just these few commands that we have already learnt we can go a long way towards writing programs that show the real power of BASIC and in particular the way computers can handle repetitive and logical tasks with ease.

## The Heart of a BASIC Program

Individual keywords can be strung together to make up quite long and complex commands. These make up the mechanics of your program. However over and above these the essence of most computer software lies in certain programming techniques that bring these commands to life, and structure them together in an intelligent and flexible way.

There are really three very simple concepts that lie at the heart of every BASIC program. Once these are understood, learning how any program works will just be a matter of studying the small details.

The three concepts are *repetition, logical tests & decision making* and *passing control.*

## Repetition - Loops

Computers have certain strengths that make them powerful and extremely flexible tools. In some ways they are also rather stupid; they have to be told everything that they are expected to do. On the other hand they are able to perform these required tasks with great speed and, at least in theory, they will be able to repeat them *ad infinitum* without ever getting bored and making a mistake.

One of the most important types of command we have available in most computer languages is known as a *loop*. This is the means by which we can make sure that commands can be repeated as many times as necessary without having to type them in over and over again.

The most common type of loop seen in BASIC programs is the FOR...NEXT statement. This works by letting you set up a simple numeric variable as a counter which controls the number of times the following instructions are to be repeated. The NEXT command signals the end of the loop and tells the computer to move the counter onto its next value. This is what it can look like in practice:

```
FOR x = 1 TO 6
   PRINT "Hello"
NEXT x
```

The result will be:

```
Hello
Hello
Hello
Hello
Hello
Hello
```

Try it out.

You can also specify the size of the jumps that the counter makes. Edit the first line to read:

```
FOR x = 1 TO 6 STEP 5
```

Do this by placing the cursor at the end of the first line and typing STEP 5.



You should then see the results:

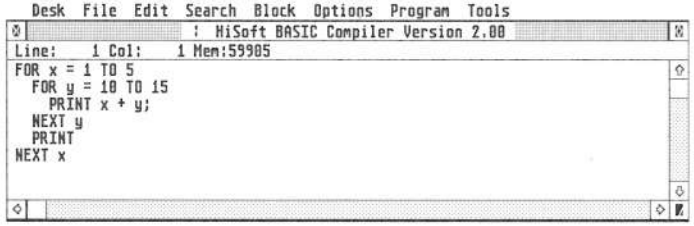
```
Hello
Hello
```

The counter values can of course be set by other variables. Consider this short program:

```
CONST start = 4, finish = 12
jump = 2
FOR x=start TO finish STEP jump
   PRINT x
NEXT x
```

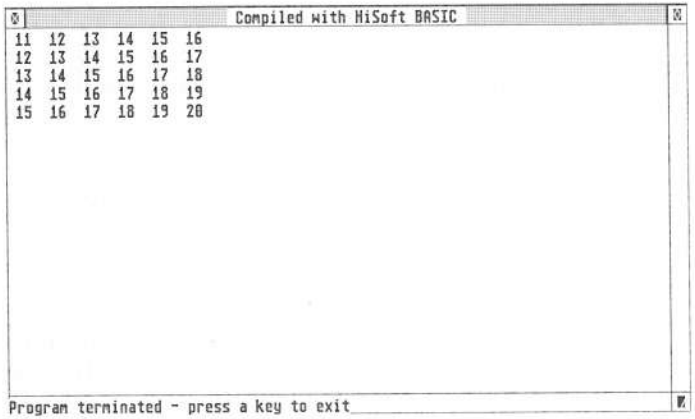
We have introduced CONST to define one or more constant values and to give them names. This is very useful for writing easy-to-understand programs and you are encouraged to use CONST whenever you have values that will not change over the life of your program. Once defined, you cannot change the value of a named constant.

One particularly useful feature of the FOR loop is that we can run one loop inside another using the technique known as *nesting*. The following illustrates this:




*Nested loops*

which produces the following output on the screen:




*A table using nested loops*

The second loop runs through in its entirety every time the first loop executes just one step. Note that we have used indentation (i.e. adding spaces or tabs at the beginning of the line) to make the logic of the program more obvious. It is good programming practice to use indentation for statements within loops.

In fact HiSoft BASIC allows you to omit the variable name in the NEXT statement as in:

```
FOR x = 1 TO 10
   PRINT x
NEXT
```

In this case the computer understands that you are referring to the counter variable x. Wherever an unspecified NEXT statement is found, the compiler will assume that it belongs to the last FOR in the listing.

However, we do *not* advocate this omission of variable names with NEXT as it can cause unnecessary confusion and bugs.

Loops are very useful things; they allow you to use the power of the language to automate all sorts of tedious tasks and achieve remarkable effects.



If that was all there was to programming, our programs would be very mundane and predictable, doing essentially the very same tasks every time they were run. To make software more sophisticated and useful we have to introduce something else, the element of choice. Programs must be able to make decisions based on the information they have been presented with, ask the user what they are to do next and act on the reply. They must then produce different output in response to the value of certain variables.

## Decision Making

Almost every program you write will use the ability of the computer to rapidly and repeatedly make logical tests and decide upon what action it should do next based on the results. A word processor decides whether to delete a word or insert a letter based on a test of what keys are being pressed; a space invaders game decides whether to 'fire a missile' based on what is happening to the joystick - it knows whether to blow up an alien based on a test of the relative position of this missile on the screen and so on.

There are two main statements that are used to make logical tests within BASIC. These are: IF...THEN...ELSE and SELECT CASE.

IF...THEN...ELSE statements are virtually self-explanatory. They take the following form:

```
IF something is true THEN
   do this
ELSE
   do this instead
END IF
```

This really is no different to any (logical) decision we make in everyday life - the computer is just making its own assessment of the current state of affairs and then deciding which instructions to follow subsequently.

The END IF command may be unfamiliar to people who have used older versions of BASIC.

HiSoft BASIC allows each set of instructions following the various parts of the structure (IF, THEN and ELSE) to cover several lines of the listing. END IF (note the space between END and IF) is therefore necessary to signal when all of the commands associated with the IF...THEN...ELSE statement have finished, and where the remainder of the program is to continue.

It is possible to omit the ELSE part of this structure, which is useful when you want the program to do something if a certain condition is true, but to carry on as normal otherwise.

IF...THEN statements can be nested within one another to increase the range of choices using the keyword ELSEIF e.g.

```
IF condition one is true THEN
   do instruction one
   ELSEIF condition two is true THEN
      do instruction two
   ELSEIF condition three is true THEN
      do instruction three
   ELSE do instruction four
END IF
```

As you can imagine it can get rather complicated for you to follow the argument and feel confident that the logic used really is correct but clear use of indentation helps greatly. Examples of IF statements follow shortly.

A structure that can often be used as an alternative to IF is the SELECT CASE structure which is ideal for allowing the computer to make one from a very wide selection of choices depending on the results of the logical test it has made. Although the exact syntax of the command when you use it will be rather different, you can think of SELECT CASE as representing a situation like (in plain English):

```
SELECT from the following
    In the CASE where this test is true
      do this
    In the CASE where the this test is true
      do this instead
    In the CASE where this test is true
      do something completely different
    In the CASE where something ELSE is true
      do this
END SELECT
```



(The END SELECT command is essential to let the computer know that it has reached the end of the instructions relating to the last CASE).

Before we can go on and give working examples of these commands and their correct syntax, it is important to illustrate exactly how the computer can go about making logical tests.

## Logical Tests

Computer logic resolves all things down to a simple test of 'true' or 'false'. These two possible conditions can be made to represent almost everything you wish, from whether a switch is on or off to whether your bank account is in the black or in the red but, in the end, all the *computer* cares about is whether the test you have requested results in a 'true' or a 'false' result.

BASIC uses the values of 0 for FALSE and -1 for TRUE. Why this is so would take an explanation of how BASIC calculates, and how it holds numbers in memory. We will be tackling these issues in more depth later but if you are ready for them now you have opened the manual at the wrong page.

The main point for beginners to bear in mind is that, while the correct explanation of how computer logic works sounds both confusing and highly unlikely, in practice using it is a remarkably simple and obvious process.

The use of logical tests in BASIC hinges around the following conditional symbols, known as *relational operators*, which express the criteria by which the computer is to determine whether the test is 'true' or not.

| | |
|---|---|
| > | greater than |
| < | less than |
| = | equal to |
| <> | not equal to |
| >= | greater than or equal to |
| <= | less than or equal to |
| == | almost equal to |

The last option, ==, will come as a surprise to anyone used to other versions of the language. It is used is to allow text comparisons that ignore any differences in the case of the two strings, and to allow comparison between two floating point numbers - which will be explained later - to ensure that a match is made despite small rounding errors during calculation.

It doesn't take a genius to figure out that the comparisons these make must be between two variables, or a variable and a constant - testing between two constants is pointless as it will give the same result every time (although this is occasionally useful to force a result).

We can now look at a typical IF...THEN statement - Close your existing source code window and type this in:

```
INPUT "What is your income this year"; income
INPUT "What will be your costs"; costs
IF income > costs THEN
   PRINT "Hooray!"
   PRINT "We're in the money!"
ELSE
   PRINT "Start packing your bags!"
   PRINT "I'll book the boat to South America."
END IF
```

Providing different values for the profit and costs will show exactly how the computer's output reflects the result of the test. Try putting a FOR...NEXT loop round the outside of the program to try out 5 different sets of profit/cost ratios.

Note that, if you use multi-line IF statements, the IF, ELSE and END IF must be on lines by themselves.

Here is an example using a SELECT CASE statement to show one version of the correct syntax. To maintain compatibility with as many versions of BASIC as possible this command structure has been implemented in a very flexible way and you are referred to the *Command Reference* section for a full list of the options.



```
INPUT "A number, please";a
SELECT CASE a
   CASE = 12
      PRINT "Number = 12"
   CASE > 20
      PRINT "Number is greater then twenty"
   CASE <= 4
      PRINT "Number is 4 or less"
   CASE ELSE
      PRINT "Number hasn't met any of my conditions"
END SELECT
```

To extend the usefulness of the above relational operators we have the keywords AND, OR, XOR, EQV and IMP. These are used to compare two separate tests and to make judgements about their relative validity.

The first two are again very easy to grasp as in the following examples:

```
IF income > costs AND tax < profit THEN
   PRINT "Hooray!"
END IF
```

The expression will only evaluate as 'true' if *both* conditional tests are themselves true.

```
IF income > costs OR tax_rebate > 5000 THEN
   PRINT "Hooray!"
END IF
```

The expression in line 1 will evaluate as 'true' if one or other or both of the tests are themselves true.

XOR is a trickier one to define as it has no direct comparison in the English language. It essentially stands for eXclusive-OR which means that the expression will evaluate as true if one or the other half is itself true but *not* if they are both true together e.g.

```
IF income > costs XOR deficit < max_tax_loss THEN
   PRINT "Hooray!"
END IF
```

EQV, short for EQuiValent, is used when two tests are to be made which need to give the same result, whether the result is true or false i.e.

```
x = 1 : y=2
IF x = 10 EQV y = 230 THEN PRINT "Yep"
```

will print Yep because both comparisons give an equivalent result i.e. 'false'.

IMP stands for 'is IMPlied by' This is without question the most abstract of the logical tests. To complicate matters even further it is the only logical operator that puts a strong emphasis on the order in which the two subtests are listed. It can best be understood as a measure of whether or not the result of the second test can be implied by, or concluded from, the result of the first test.

The rules are that:

- a TRUE result can be concluded from a preceding TRUE result;
- a FALSE result can be concluded from a preceding FALSE result;
- a FALSE result can be concluded from a preceding TRUE result;
- a TRUE result cannot be concluded from a preceding FALSE result.

The premise is that it is possible to draw the wrong conclusions from correct assumptions, but it is impossible to draw the correct conclusions from wrong assumptions - try these out:

```
x = 1 : y = 2

IF x<2 IMP y<3 THEN PRINT "Yep" ELSE PRINT "No"

IF x=2 IMP y=10 THEN PRINT "Yep" ELSE PRINT "No"

IF x=1 IMP y=10 THEN PRINT "Yep" ELSE PRINT "No"

IF x=2 IMP y=2 THEN PRINT "Yep" ELSE PRINT "No"
```

The implications of all the above logical operators can be reversed by the use of the keyword NOT as in:

```
IF NOT (x=10 AND y=12)
```

and so on.

## Testing Text

It is of course equally possible to make logical tests on text data. The relational operators we use to express the test are the same as with the numeric examples given above but the way in which the computer evaluates whether the result is TRUE or FALSE is rather more complex.



Possibly the simplest to start with is to test whether two text strings are equal (=) or not equal (<>). Obviously this example:

```
test$ = "FRED"
IF test$ <> "BILL" THEN BEEP
```

is straightforward.

But what about operators such as greater than (>) or less than (<)? How can we say that one word or letter is 'greater' than another?

To understand the answer we will have to make a diversion into looking at the way computers store letters in their memory.

Despite the way they look on screen, computers store letters in their memory as numbers. All modern computers stick to an American convention known as ASCII (which incidentally stands for American Standard Code for Information Interchange) whereby each letter has a certain number associated with it. By standardising things in this way it has made it possible for computers to talk to each other and send text such as Hello Jim without it coming out as ghJJm KPhZz.

A list of the characters used by your computer, together with their relative ASCII numbers, will be found in *Appendix D* of the *Technical Reference* manual. Note that all lower case letters have higher numbers than all the upper case ones. The digits (0-9) come before both. The letters with a value of 0-31 in the ASCII table look a bit odd - these are actually special control characters. Sending these, or a combination of them, to the screen or printer often produces a response such as changing some characteristic of the display, or type style, or moving the cursor.

In a logical test, the first character of each of the two strings that are being compared will be tested - if one has a higher ASCII number than the other it will be regarded as 'greater'. If both characters have the same ASCII number the computer will move on to the next character in both strings.

If both strings are the same for all of their common length except that one is longer than the other, the longer one will be regarded as greater rather than the shorter string.

Note also that even a blank space is a character and it comes before all others (value 32 in the ASCII table). A leading space in some of the data is a common way in which text comparisons can become completely confused, and it can be a very difficult problem to spot.

Perhaps it all sounds a little complex, but the encouraging thing is that in practice it is easier than it looks. Common sense will usually tell you what the outcome of any test between two strings will be. The only time it is likely to be as clear as mud is when punctuation and unusual characters such as Ü, Ø, and ß are compared, as these tend to occur in unpredictable places in the standard ASCII character set.

The relational operator == is particularly valuable when making text comparisons, as it can be used to ignore any differences between the case of the letters being compared e.g.

```
INPUT "Are you happy"; ans$
IF ans$=="Yes" THEN PRINT "I'm glad!"
```

Will give a glad message if you respond with `YES`, `yes`, `Yes` etc. but not if you answer `No`.

## Logical Loops

We have so far seen how to get the computer to execute a series of instructions a set number of times as a loop. We have also seen how to make it decide, from a series of options, what it is to do next, based on a logical test.

The situation often arises where the user simply cannot predict how many times a certain loop will be required to execute and it would be ideal if we could get the computer to use its logic to decide this for itself whilst running. HiSoft BASIC provides several methods of doing just this, based on special loops that continue to execute until a logical test tells them to stop.



(Note that many inferior BASICs which lack these facilities sometimes forced the user to insert a logical test within a `FOR...NEXT` loop. This test would force an unannounced jump from within the loop and end its execution when the true result was reached. Most authorities would agree that this is bad programming practice although it was often the best way of getting the job done under the circumstances. Unlike some languages, HiSoft BASIC can handle such 'jumps' without complaint, but the programs you write will be much clearer if you avoid doing so and use the following structures.)

The three logical loops that we can use are known as the `WHILE...WEND`, `REPEAT...END REPEAT` and `DO...LOOP` statements.

The first one works like this.

### The WHILE loop

```
WHILE conditional test is true
      do this
      do this
WEND
```

The `WEND` keyword simply signals the end of the instructions that are to be included within the loop.

Here is an example:

```
INPUT "What is your bank balance"; balance
income = 50
month = 1
WHILE (balance>0) AND (income>49)
   PRINT "What is the income for month"; month;
   INPUT income
   PRINT "What are the costs for month"; month;
   INPUT costs
   balance = balance + (income - costs)
   PRINT "New balance is"; balance
   month = month + 1
WEND
PRINT "It's time to look at getting a proper job!"
```

Note that, in order to ensure that the `WHILE` loop is executed at least once, we have had to initialise `income` to 50 (>49) before the beginning of the loop.

In fact, it would be better, therefore, to use a different type of loop, the DO loop, for this example and we shall now see why.

## The DO loop

The DO loop actually has several permutations which make it a very much more flexible construction than the WHILE loop. In fact it even has an option that will completely simulate the WHILE...WEND command. The various permutations include:

```
DO
      this instruction
      that instruction
LOOP UNTIL condition

DO UNTIL condition
      this instruction
      that instruction
LOOP
```

Can you see that these constructs are similar to a WHILE loop? Are there any differences?

Yes ... a DO UNTIL...LOOP executes the instructions in the loop until the condition becomes true whereas a WHILE loop executes until the condition becomes false. Also DO...LOOP UNTIL checks its exit condition at the end of the loop, not at the beginning. Thus, in the previous example, since we did not know the value of income at the start of the loop, perhaps we should have used a DO...LOOP UNTIL loop, not a WHILE loop - can you re-write it?

There are other permutations of DO:

```
DO WHILE condition
      this instruction
      that instruction
LOOP

DO
      this instruction
      that instruction
LOOP WHILE condition
```

As we said above, the logic of the WHILE comparison is opposite to that of the UNTIL option:

```
DO UNTIL x = 10
```



and

```
DO WHILE x <> 10
```

will produce exactly the same effect.

The most general form of the DO loop is:

```
DO
      this instruction
      that instruction
LOOP
```

This command structure will continue to run and run unless there is a logical test inserted somewhere within it terminating in an EXIT DO or EXIT LOOP command; these two are interchangeable.

It is again possible to nest such loops several times, and indeed to nest one type inside another.

It is also possible to have conditional tests at both ends of a DO...LOOP.

## The REPEAT loop

The third type of logical loop is the REPEAT...END REPEAT structure. This has the most in common with the final form of the DO...LOOP that we have just seen, in that there is no inbuilt way for the loop to terminate beyond the inclusion of one or several EXIT statements.

The difference between the two types of loop hinges on the fact that each REPEAT loop you create can be given its own name. Several such loops can be nested culminating in a series of logical tests with EXIT statements. Any EXIT statement can itself refer to the name of a given loop, but not necessarily the one that was most recently defined. In other words you can EXIT a named outer loop from within an inner one; very useful for aborting some task if a catastrophic error occurs.

We advise the use of DO...LOOPs wherever possible since they have a much more coherent structure - try to avoid REPEAT loops unless absolutely necessary.

## Passing Control Within a Program

The effect of logical tests is to force the program to make a decision about which lines of the program are to be executed next. In other words the flow of the program is passed from one section to another.

Most large programs contain sections that are only executed as and when certain conditions are met. A space invaders game for example will have a GAME OVER message that will only be displayed when the program has realised that you have had all of your missile bases destroyed. As long as you can continue playing unscathed, control of the computer will never pass to the end-game section of the program.

In the above examples the logical tests have passed control to one or two lines of the program, for very simple results. However there is no reason why there should not be enormously long and complex sections of the program that are accessed in each case.

There are two distinct mechanisms that we can use to cause the computer to move to another section of the program. These are *jumps* and *subroutine calls*. The difference between the two is that when the computer jumps to a different part of the program it does not bother to remember where it came from and it is not able, and does not try, to get back there unless you explicitly tell it where to go. By contrast, subroutines and sub-programs are self contained sections of the program that fulfil a specific task and then automatically return back to the next instruction following the one that first called them.

In older versions of BASIC, versions that required the user to begin each line with a number, jumps were made by simply entering GOTO line number as in the following example.

```
10 LET x = 1
20 LET y = 20
30 LET x = x + 1
40 IF x = y THEN BEEP ELSE GOTO 30
```

This example also shows how very primitive logical loops can be created - the program will jump between lines 30 and 40 until the test is passed as true.



Although it is capable of running the above program, HiSoft BASIC does not force the programmer to use line numbers. But in that case how would we signal to the program when it is to make a jump, and where it is expected to go to?

The answer is through the use of line *labels*, which are essentially names that we give to chosen lines in the program, usually lines that start a certain routine. By making a jump to one of these names the same effect is achieved as with a line number jump but the listing is made much more readable. Line labels are entered by just typing the required name followed by a colon.

```
maingame:
   .
   .
IF missile_base = 0 THEN GOTO endgame
   .
   .
endgame:
BEEP
PRINT "Game over."
```

In older versions of BASIC, subroutines were similarly accessed by a call to a certain line number which was to be the beginning of the routine itself. The keyword that makes this call is GOSUB. Here is an example:

```
10 INPUT "What is your bank balance"; x
20 IF x >0 THEN GOSUB 100 ELSE GOSUB 200
30 INPUT "How much will your bills be this week"; z
40 IF z > x THEN GOSUB 200

.100 BEEP
110 PRINT "There will be no bank charges"
120 RETURN

200 BEEP
210 PRINT "There will be bank charges"
220 PRINT "unless you pay some money in"
230 INPUT "How much can you pay in"; y
240 IF y < x THEN PRINT "Not enough I'm afraid"
250 END IF
260 LET x=x+y
270 RETURN
```

The keyword RETURN is essential to signal the end of the current subroutine, otherwise the subroutine would carry on *ad infinitum*.

The above example appears slightly laboured because in very short programs it can be difficult to justify the use of subroutines as separate entities from the main lines of the listing. In long and complex programs however they perform several invaluable roles.

Firstly they allow the programmer to divide up his work into a series of logical modules, each one of which can be worked on, altered, tested and debugged in isolation. As well as making things easier for the author of the program, they also make the listing very much easier to follow and alter at a later date.

As an extension of this philosophy, once a subroutine has been perfected it can be made part of a stored 'source library' of routines which can, at any stage, be slotted into new programs that are under development. For example there may be a subroutine that plays a tune which you can use in any game you write, or you may have some lines which place a menu of options on a screen, or which reads data from a disk file.

Another important use for them is to make the actual listing of your programs more compact and economical. Any program routines that are used several times within the same listing need only be typed once, and they can then be accessed by the appropriate subroutine call as often as necessary.

HiSoft BASIC will of course allow the use of simple subroutines as described above but in addition, as with GOTO, it is possible for the routine to be called by a name rather than by its line number, which of course makes the listing easier to follow.

Avoiding numbers also makes it logistically simpler to merge many different library procedures together into a new program.

Another improvement is that the RETURN command can be followed by a line number or label that will allow the program flow to be passed to yet another part of the listing.



## A Better Way; Sub-programs

As well as standard subroutines, there is a much more satisfactory way of organising programs into modules which are called *sub-programs*.

Sub-programs have three advantages over subroutines, which may not be appreciated by many programming beginners but will become more and more important as your skills increase together with the complexity of your programs. These are:

- local variables
- parameter passing
- recursion

Each of these will be explained as we progress with the tutorial.

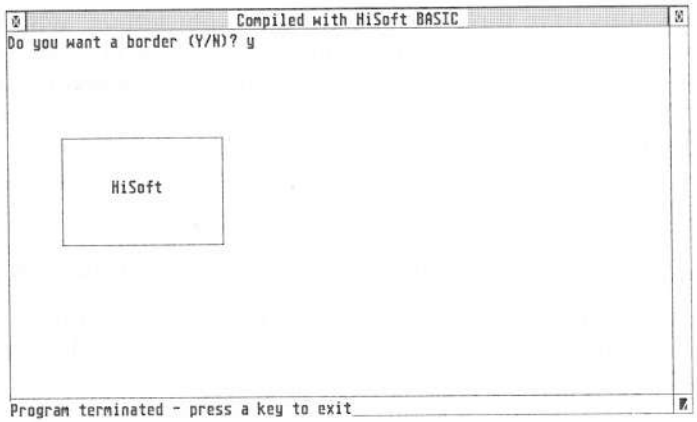
The keywords used for defining sub-programs are SUB and END SUB. Once defined they can be CALLed from anywhere else in your program listing.

If you have been paying attention until now, you should find it fairly easy to decipher the following example...

```
Desk  File  Edit  Search  Block  Options  Program  Tools
                 ¦ HiSoft BASIC Compiler Version 2.00
Line:   18 Col:    1 Mem:59675
Mainprogram:
LOCATE 9,13
PRINT "HiSoft"
CALL box
STOP

REM Sub-program to optionally box some text
SUB box
  LOCATE 1,1
  INPUT "Do you want a border(Y/N)"; Ans$
  IF Ans$="Y" OR Ans$="y" THEN
    LINEF 50,50,50,50
    LINEF 50,100,200,100
    LINEF 200,100,200,50
    LINEF 200,50,50,50
  END IF
END SUB
```

*Using a sub-program ...*

Try typing this in and running it. It has deficiencies; it only works properly in ST medium resolution; the sub-program should be passed, as two parameters, the point at which the text has been drawn and it should then work out where to put the box ... but you see the idea.




*... to get HiSoft in a box!*

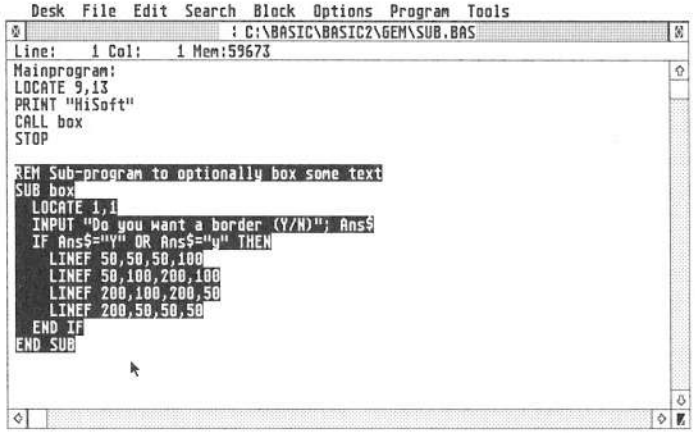
Note the `LOCATE x,y` statement, after `Mainprogram:`, which positions the cursor at row `x`, column `y` within the window, so that the next `PRINT` takes place there. The origin is 1,1 not 0,0 as it is for graphics statements.

Here's a chance to show off a useful feature of the editor; you often want to use a sub-program or a function in many different programs - after all, that's one of the main reasons for coding them in the first place. You could group all your sub-programs together in one program, save it and 'include' it in whichever program was going to use the sub-program, function etc.

However, there is an easier way that you may wish to use sometimes - simply cut the sub-program from one window and paste it into another, here's how:

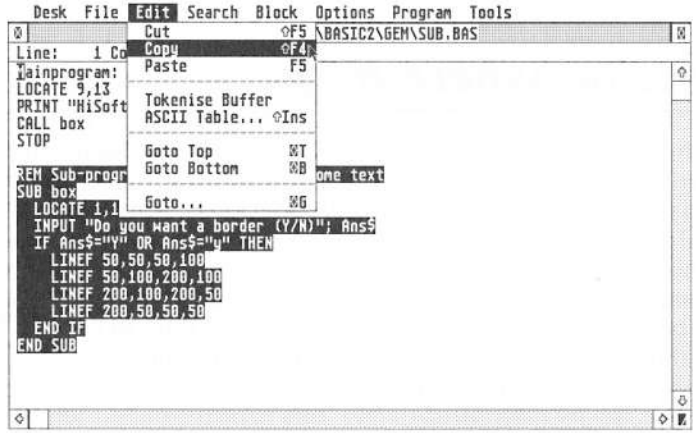
Position the cursor at the start of the `REM` statement, click and hold the button down. Now drag the mouse until you have marked all the text up to the end of the `END SUB` and release the mouse button.






*Marking a block*

Now select Copy from the Edit menu; this copies the marked text into memory.




*Cut, copy and paste*

Now open a new window (say, Alt-3) and select Paste from the Edit menu. The complete sub-program is copied into your new window, ready for use.

Back to sub-programs ...

It is not uncommon to find well written modular programs where the main body of the listing is no more than a series of calls to different subprograms as in the following:

```
CALL start
CALL input
CALL output
CALL end
```

You can invoke a sub-program called fred either by using CALL fred or simply using the name of the sub-program, fred. In this way routines can be executed just by using their names, almost as if they were special BASIC keywords that you have written yourself. The *Command Reference* section gives more details. However, we would like to stress one point here:

`CALL fred (john)` is *not* the same as `fred (john)`

When using CALL, you *must* enclose the parameters within parentheses and they are then passed as *variable parameters* (see below). However, when *not* using CALL, enclosing parameters in parentheses forces the parameters to be passed by *value*. Don't blame HiSoft. It's historical!

More of sub-programs later.

## Choosing where to go

A useful command structure that allows us to program jumps to different routines in a very concise way is ON x GOTO/GOSUB. In this case x is a numeric variable that can contain a number generated by the program, or entered by the user. The command sequence tests the value of x (or rather the integer value of x - see later) and will jump to the xth entry in a list of subroutine names, line labels or line numbers that can follow the GOTO or GOSUB command. This list can be up to 56 items long and can mix line or subroutine names and numbers.

```
Mainprogram:
REM Other lines of the program

PRINT "Data listed on screen or printer?"
PRINT "Screen.......1"
PRINT "Printer......2"
DO
   BEEP
   INPUT answer
LOOP UNTIL (answer = 1) OR (answer = 2)
ON answer GOSUB screenprint, paperprint
```



Again the example looks like a rather longwinded way to achieve a simple effect but in complex programs this command structure is invaluable. The chosen variable in your own program can be used as an indicator (in computer jargon: a flag) of what routines have been accessed or what choices have been made by the user and the program response tailored to fit.

It is not possible to call sub-programs with this command; to do this we recommend you use a SELECT CASE structure with CALLs which would look like:

```
Mainprogram:
REM Other lines of the program

PRINT "Data listed on screen or printer?"
PRINT "Screen.......1"
PRINT "Printer......2"

DO
   INPUT answer
   SELECT CASE answer
      CASE 1 : CALL screenprint
      CASE 2 : CALL paperprint
   CASE ELSE BEEP
   END SELECT
LOOP UNTIL (answer = 1) OR (answer = 2)
```

In general, we would advocate the use of CASE over ON GOTO/GOSUB since it normally leads to clearer programs.

## Sub-programs

As we have said above, sub-programs are much more powerful than old-fashioned subroutines and we encourage you to use them whenever you can.

Sub-programs have the ability to define and use variables that are only valid within the routine itself, but which have no effect on other variables of the same name used elsewhere in other routines or within the main program body. These are what are known as *local variables*, because they are only recognised and used within one routine. They make it immensely easier to write and test procedures as independent modules without worrying about compatibility with others that do different tasks.

Say that we want to write a sub-program that takes a text string and prints it ten times on the screen. Our sub-program may be of the following form:

```
SUB Mul_Print
   FOR x = 1 to 10
      PRINT A$
   NEXT x
END SUB
```

Do not try entering this routine just yet - we will see shortly that it is not quite finished. Firstly, note that we have introduced a variable, x, that is used in the FOR...NEXT loop as the loop counter - it is obviously only needed while the loop is being executed and is therefore a prime candidate for being a local variable - we make it such by adding the statement STATIC x after the SUB definition. We have used the keyword STATIC to declare x as local to this sub-program - it will not be available outside the sub-program (but see SHARED later).

> If you are peachy-keen you may have noticed the keyword LOCAL in the *Reserved Words* list or the *Command Reference* and you may be wondering why we have not declared x as LOCAL. This is because STATIC x is more efficient on memory and runs faster - LOCAL x introduces a new variable every time it is encountered whereas STATIC x creates only one variable which is re-used. The main occasion when you might want to use LOCAL rather than STATIC is if you have a sub-program or function that calls itself and you need independent local variables within each call.

Also, we have written the sub-program using the text variable name of A$. However to satisfy the requirements of flexibility and independence from the main program variables we have to find a way to tell the sub-program exactly what A$ represents when we call it. The situation may be that we may not have used that variable name in the main program, or even that we have used it for something completely different.



Let's write some code that might call Mul-Print:

```
Main_Program:
A$ = "Banana"
B$ = "HiSoft = HIgh quality SOFTware"
PRINT "MENU"
PRINT "1. "; A$
PRINT "2. "; B$
INPUT x
SELECT CASE x
   CASE 1 :  choice$=A$
   CASE 2 :  choice$=B$
   CASE ELSE  choice$="Not given"
END SELECT
```

The next thing we want to do in this program is to make a call to the Mul_Print sub-program. We need to let this sub-program know that we want it to print choice$ rather than A$, even though we have used the name A$ in the actual Mul_Print routine. The SUB was possibly written months before and stored in a library.

Our old type of subroutine detailed above will have been unable to allow for this necessary degree of flexibility. However when using sub-programs we can do it with ease thanks to a technique known as *parameter passing*.

To use this we first have to alter the first line of our sub-program to contain not only the name of the procedure, but also the parameters it will expect to be given from the main program.

In our example this would be:

```
SUB Mul_Print(BYVAL A$)
STATIC x
   FOR x = 1 to 10
      PRINT A$
   NEXT x
END SUB
```

> The BYVAL keyword in the definition of the sub-program indicates that the A$ variable is to be always passed by value, not by reference, which is the default. Unless you need a parameter to be variable (passed by reference), we would encourage you to use BYVAL to make it passed by value since this is more efficient on memory and also produces faster-running programs. See below for more detail.

When we actually make the call to the sub-program from the main program we have to also include in that line the appropriate variables, numbers or text strings that we want each passed variable to represent.

In our example this line would be:

```
Mul_Print choice$
```

Not surprisingly the parameters that are passed must match the parameters expected by the sub-program in number, type and order but they need not match with regard to the actual names used.

The complete program would be:

```
SUB Mul_Print(A$)
STATIC x
   FOR x = 1 to 10
      PRINT A$
   NEXT x
END SUB

Main_Program:
A$ = "Banana"
B$ = "HISOFT = HIgh quality SOFTware"
PRINT "Menu"
PRINT "1. "; A$
PRINT "2. "; B$
INPUT x
SELECT CASE x
   CASE 1
      choice$=A$
   CASE 2
      choice$=B$
   CASE ELSE  choice$="Not given"
END SELECT
Mul_Print choice$
```



Not particularly useful, but it illustrates the point. Note how the sub-program is defined before the main program - this is good programming practice, but not essential.

Now try editing the main program to include a second option for choosing how many times the string is to print. The `Mul_Print` sub-program will have to have a second parameter entered into the opening line, say `Mul_Num` which will be used in the the line:

```
For x = 1 to Mul_Num
```

Remember to put a second parameter in the call to the sub-program to pass the value for this variable that has been selected by the user.

By default HiSoft BASIC assumes that all variables that are declared or used by a sub-program are STATIC variables i.e. are local to the sub-program and will not affect or be affected by the values of variables of the same name elsewhere in your program. STATIC variables are zeroed when your program is first executed but are then left alone by HiSoft BASIC - their values are held static until you change them within your program.

However, even though STATIC is the default, it is advisable to explicitly declare any local variables as STATIC (or LOCAL) since this improves the readability and maintainability of your program. Also, if you have variable checks on (a compiler option), an error will be reported if you use variables in a sub-program or function that have not been declared.

However there are situations where it would be valuable for the computer to create a new variable each time a sub-program is called. If you want HiSoft BASIC to create a new variable for each sub-program call then you should declare the variable as LOCAL, not STATIC.

There is also a way of letting a sub-program act on and even change the values of variables that are used by the main program and without these having to be passed to the subprogram as parameters. We do this by stating, before they are declared or used by the sub-program, that a certain variable name is to be SHARED.

As you will be able to deduce from the example programs we have used so far HiSoft BASIC does not force you to define exactly which variables belong in which categories. The compiler is able to deduce the effect that you are trying to achieve unless there is a mistake in your own logic. For that very reason it is good programming practice to work out and specify exactly which variables belong to which types in your program.

```
SUB testproc
SHARED x,y
   PRINT x, y , z
   x = x + 10
   y = y + 10
   z = z + 10
END SUB

Mainprogram:
   x = 10 : y = 20 : z = 50
   PRINT x, y, z
   testproc
   PRINT x, y, z
```

This will produce the following output:

```
10    20 50
10    20 0
20    30 50
```

which illustrates that x and y were usable, and capable of being changed, by the sub-program even though they were not passed as parameters. The third variable, z, was in fact treated as two entirely separate creatures because, by default, it was STATIC to the sub-program. x and y were created by, and belong to, the main program unless SHARED; z belongs exclusively to the sub-program.

> Remember, the default situation is that every variable used in a sub-program is regarded as STATIC unless specified otherwise. The use of variables that are SHARED between sub-programs and the main program should be treated with extreme caution since it is easy to introduce obscure bugs using SHARED variables - see the SHARED variables section in the *Concepts* chapter for an example.



SHARED variables are not the only means by such a sub-program can pass information or changes back to the main program body and this brings us on to a discussion about *Value Parameters* and *Variable Parameters*.

## Value and Variable Parameters

There are two different ways that you can pass parameters to and from sub-programs with HiSoft BASIC; essentially you can either pass just the *value* of the parameter to the sub-program or you can pass a *reference* to the parameter. In the latter case the parameter is called a *variable parameter* because the sub-program can actually modify the value of the variable since it knows where to find it - all parameters passed to sub-programs are, by default, variable parameters. This provides another mechanism, along with SHARED variables, for sub-programs to communicate with the main program and each other. For example:

```
SUB Strip_Spaces(a$)
STATIC b$,i,j
b$=""

REM Find first non-space character
i=0
DO
   i=i+1
LOOP UNTIL MID$(a$,i,1)<>" "

REM Now copy rest of string to temporary string
FOR j=i to LEN(a$)
   b$=b$+MID$(a$,j,1)
NEXT j

REM Copy back to passed string
a$=b$
END SUB

test$="    HiSoft"
Strip_Spaces test$
PRINT test$
```

Try this for yourself; it shows the use of a variable parameter to strip the leading spaces from a string variable. We have used a few string functions (like MID$ and LEN$) that you may not have seen before - don't worry, they will be explained later.

In fact, there is a built-in function called LTRIM$ which will strip leading spaces for you - see the *Command Reference*.

HiSoft BASIC allows you to overide the default that a parameter is a variable parameter - simply enclose the parameter in parentheses in the call to the sub-program e.g. if we had used `Strip_Spaces(test$)` above, `test$` would have remained as `HiSoft`.

If, instead, you use CALL to invoke the sub-program and you want the parameter passed by value when it has not been declared as such in the sub-program definition then, again, enclose the parameter in parentheses, for example:

```
CALL Strip_Spaces ((test$))
```

The alternative to variable parameters is the *value parameter* where only the value of the variable is passed to the sub-program and the variable itself cannot be modified. To indicate that you want a variable to be passed by value, you should precede it with the word BYVAL or VAL in the sub-program definition.

For example:

```
CONST FALSE=0

SUB Factors(BYVAL Number)
STATIC i
   FOR i=2 TO Number/2
      IF Number MOD i=0 THEN PRINT i
   NEXT i
END SUB


DO
INPUT i
Factors i
LOOP UNTIL FALSE
```

There is no *need* to make `Number` a value parameter in this example since it is not modified within the sub-program - but value parameters are processed considerably faster than variable parameters and, in a calculation-intensive (albeit simple) sub-program like this one, speed can be of paramount importance.



So it is advisable to define your parameters as value parameters unless you need them to be otherwise.

Note also the use of CONST to define a constant FALSE that can then be used with the DO...LOOP to loop forever. Type in this program and run it - when you get bored hold down Ctrl-C to break out of the program.

Now for a little fun...

## Recursion

Sub-programs do of course have the ability to call other sub-programs in turn, or even to call themselves if necessary. The level of complexity of such inter-related calls can be enormous, yet the resulting listing will remain obvious and easy to follow because the procedure name can almost be regarded as a brand new keyword that does a certain job - contrast that with a similar situation using line numbers.

The ability of a sub-program to call itself is known as *recursion*. This is an enormously useful and powerful programming tool that opens up whole new worlds of opportunity and in doing so flies completely over the head of 90% of computer users.

Unfortunately the programs used to demonstrate the principle usually employ it to solve complex and obscure mathematical problems that are far from easy to follow. We are not intending to let you off the hook either - on the HiSoft BASIC disk you will see a version of the infamous 'Towers of Hanoi' program, called HANOI.BAS. This demonstrates the full power of recursion by solving the sort of horrific mind bending puzzle that most people cannot visualise anyway and you should certainly study it when you have finished working through this book.

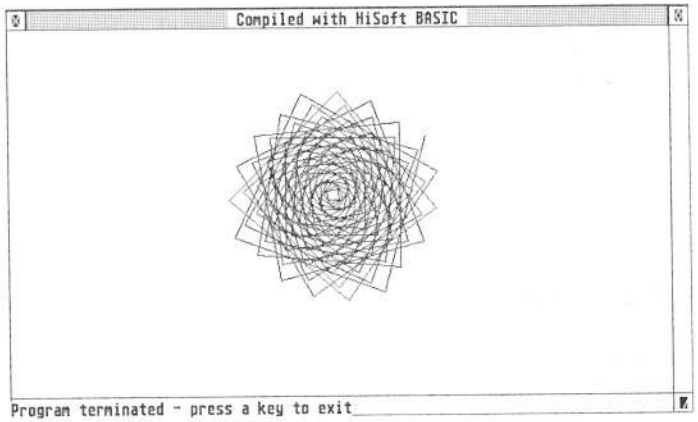
However, here is a procedure that demonstrates recursion, at least to the extent of proving that it really does work. Like many recursive problems the same or a similar effect can be achieved by the use of logical loops, but you will often find that using procedures in this way is often much more efficient in both speed and compactness of your program.

Type this in:

```
' The ForWarD sub-program. Draws a line of length
' r in the direction, dir, of the 'turtle',
' from its current position
SUB FWD(BYVAL r)
SHARED curx,cury,dir
STATIC newx,newy
' Calculate the new x and y co-ordinates
   newx=curx+r*cos(dir)
   newy=cury+r*sin(dir)
' Draw the line
   LINEF curx,cury,newx,newy
' Update the (x,y) position of the turtle
   curx=newx
   cury=newy
END SUB

' Turn the turtle through r degrees
' i.e. simply change dir
SUB RIGHT(BYVAL r)
SHARED dir
   dir=dir-r/180*3.1415926
END SUB

' Use FWD and RIGHT to draw a spiral.
' This sub-program is recursive, it calls
' itself to draw successive, longer lines
' to produce a spiral
SUB spirals(BYVAL L,BYVAL A)
   FWD L
   RIGHT A
   IF L<150 THEN spirals L+1,A
END SUB

' Initialise the turtle and draw a spiral
' You can try changing the numbers 9,95 to
' obtain different shapes

main:
curx=160 : cury=100 : dir=0
spirals 9,95
END
```



The above program produces the output shown below:




*A recursive turtle*

Beginners may feel that they have been given a fairly rough ride over the last few sub-sections so let's get back to some simpler BASIC concepts for a while.

## Functions

Functions are an extremely important group of part of HiSoft BASIC that encompass almost every type of task you can wish your computer to do, they can control graphics, text, calculation, logic, indeed there are very few programs that do not use functions of some sort in almost every line. Any attempt to define what functions do is therefore almost pointless - they can do anything and everything.

What makes a function a function as distinct from any other element of BASIC is that, once called, it has to return some form of information, text or numeric data, to the original program. Most, but certainly not all, functions need some information to be passed to them from the main program to operate on.

Because of their flexibility we will only briefly run through some of the different types of functions here, the majority of them will be covered as and when they come up under different logical headings in the tutorial and some will be left for you to discover in the *Command Reference* section.

## Mathematical Functions

We have already seen how the use of different arithmetic signs, add, divide, multiply and divide can make your computer into a kind of calculator. To extend the usefulness of these HiSoft BASIC comes with an extensive range of mathematical functions. COS and LOG are two examples:

```
PRINT COS(45)
y = LOG(x) : PRINT y
```

## Text Functions

Most text functions are designed to perform operations on existing text strings. They will therefore be covered in depth in the section *Strings and Things*.

Exceptions include two text functions that are designed to create new strings - SPACE$ and STRING$.

```
A$ = SPACE$(10)
```

will create a new string of ten spaces, which can be printed or assigned to a variable.

STRING$ can be used to produce a string of specified length made up of a chosen character repeated:

```
x = 23
PRINT STRING$(x,"*")
```

which will print:

```
**********************
```

## Miscellaneous Functions

There are a wide selection of functions available that fit into no neat category, but just perform a useful job. Two simple examples are DATE$ and TIME$ which return the appropriate values from the computer for display within your own programs.



```
PRINT DATE$
```

may give the answer

```
22-04-1992
```

Remember that a function, by definition, must return some sort of value or data to the program that called it.

It is also possible to use the above two keywords as command statements rather than as functions as in the line:

```
DATE$ = "01-01-1992"
```

which assigns a new date value to the system clock. The difference between these two sides of the same keyword is at the same time trivial and crucial.

## User Defined Functions

HiSoft BASIC has a very generous smattering of functions designed to provide you with a wide selection of useful tools for manipulating data. However it is impossible to predict or to cater for every possible eventuality that will come up. There will inevitably be times when you will want to use functions that do not come supplied. Fortunately you can then employ User Defined Functions to construct new routines to meet your own requirements.

HiSoft BASIC differs from many other versions of the language in that it allows the user to write complex multiple-line functions. These have many similarities with sub-programs. They can have parameters passed to them (and by definition must pass a result back to the calling program line). Also, they can use local and global variables and as such can easily be stored in a pre-written library of routines.

The keywords used to set up and call a user-defined function are FUNCTION / END FUNCTION or DEF FN / END DEF / FN. As with sub-programs and logical loops the command EXIT FUNCTION / EXIT DEF can be used to trigger a jump from the function, e.g. if it has been passed a value that is outside of the range that you wish to permit.

The name that is used to define a function as in the commands:

```
DEF FNname or FUNCTION Cube
```

is also the name used to call the function using the command:

```
FNname or cube
```

and is also used as a variable, within the function definition, which is assigned the value that is to be returned to the calling program line. Note that there are essentially two ways of defining a function; using the FN terminology or, more simply, using the FUNCTION keyword; we encourage you to use the latter, more modern method.

Here is an example of a mathematical function that can be used to convert inches into centimetres.

```
FUNCTION metric(a)
   metric = a*2.54
END FUNCTION

Mainprogram:
   Inches = 10
   CMs = metric(Inches)
   PRINT Inches; "inches equals";CMs;"centimetres"
```

The following example of a text function can be used to strip trailing and leading spaces from any string. It employs one or two pre-defined text functions as building blocks; to follow these you must refer forward to the section on strings, or consult the *Command Reference* section.

```
FUNCTION strip$(A$)

   IF LEN(A$)=0 THEN EXIT FN
   WHILE LEFT$(A$,1)=" "
      Length=LEN(A$)
      A$=RIGHT$(A$,Length-1)
   WEND

   WHILE RIGHT$(A$,1)=" "
      Length=LEN(A$)
      A$=LEFT$(A$,Length-1)
   WEND

   strip$=A$

END FUNCTION
```



> The above example is useful as an exercise but HiSoft BASIC lets you strip leading and trailing spaces with LTRIM$(RTRIM$(A$))!

If you use a function, declared using FUNCTION, *before* it is defined, you must tell HiSoft BASIC that you are going to do this by using DECLARE FUNCTION name followed by its parameter list, at the front of your program. Otherwise the compiler may think that you are using an array name and not a function call e.g.

```
DECLARE FUNCTION Cube (BYVAL x)

PRINT Cube(5.6)

FUNCTION Cube(BYVAL x)
   Cube=x*x*x
END FUNCTION
```

In general, it is a good idea to include all your sub-program and function definitions at the beginning of your program.

> Note that parameters passed to user-defined functions defined using DEF FN are, by default, *value* parameters - you can force a variable parameter by including the word VARPTR before the variable in the function definition. Also local variables within a DEF FN function are automatically assumed SHARED. This is all historical and we have included this behaviour in HiSoft BASIC to be compatible with Microsoft QuickBASIC et al.
>
> Functions declared using the keyword FUNCTION do not suffer from this confusing behaviour towards parameters and local variables - they behave in exactly the same way as sub-programs; parameters are, by default, variable and local variables are, by default, STATIC. Therefore we advise you, *most strongly*, to use FUNCTION and not DEF FN, which is provided only for backward compatibility.

# More about Numbers & Text

The ways that HiSoft BASIC handles data, particularly numbers, is very much more complex than we have seen so far. This is not sadism on the part of the designers but is in fact essential (honest!).

In every day conversation we all use numbers in a variety of ways and, most of the time, other people manage to work out exactly what we mean. For example you may say to someone 'I'll be around in ten minutes' and it is understood that you could in fact be eight, twelve or even twenty minutes. Conversely if you say 'I want to collect that ten pounds you owe me' you will be rather unhappy to receive any less than that. The essential difference between the two is the precision of the figures we use and the implicit understanding of the person to whom we are communicating.

The problem is that while people are able to use their experience to interpret what you probably mean, a computer takes everything much more literally. It will expect you to turn up in exactly 600 seconds to collect your 1000 pence.

As if that was not enough, even computers, which insist that you say exactly what you mean, do not always mean exactly what they appear to say. What would you make of a calculator that claimed that two plus two did not equal four. Not much probably, but it is actually possible for this to happen.

Numbers that look quite simple to the user, such as '4', may in fact be stored as very much longer figures e.g. '4.00012'. These slight aberrations from what the user sees, or means when typing in figures, result from minute rounding errors in the way that the calculations are performed internally.

Obviously, this state of affairs can not be left to operate randomly. Small rounding errors of this sort, when added together or multiplied can eventually result in visibly different answers from that which is expected.

We therefore have different ways of expressing numbers and numeric variables in HiSoft BASIC that will control the degree of precision that you want the calculations to work to, and can force numbers to change their internal format to ensure that they achieve the desired result.



The most important distinction is between integer and floating point numbers. An integer number is a whole number, i.e. there can be no decimal fraction. By forcing the computer to use integer numbers we can ensure that all calculations produce the result that we would expect, 2+2 really does equal 4. Integers are particularly useful when performing financial calculations where it is important that the odd penny does not go astray.

Unfortunately, for unimportant reasons which have to do with the way that computers work, integer numbers usually have to fit within the range +32767 to -32768. In order to maintain compatibility with programs that run under other versions of BASIC, the HiSoft BASIC recognises and uses the ordinary integer type and the associated commands that allow variables to be assigned as integers. However this version of the language has also been extended to also allow long integers, whole numbers in the range 2147483647 to -2147483648.

Floating point numbers are any numbers that have a fractional part, i.e. there is a value after the decimal point. Again there are two different types of these numbers - single and double precision numbers.

Floating point numbers are often displayed using scientific notation consisting of a fixed point number (the mantissa) followed by a letter E and then an integer which is the exponent. To convert this notation to a normal format you have to multiply the mantissa by ten to the power of the exponent.

Single precision floating point numbers are accurate to seven digits in the mantissa while double precision numbers are accurate to 16 digits in the mantissa. The exact ranges for these numbers are given in the *Concepts* chapter.

For small number values, where there is no ambiguity in the precision, numbers will be displayed in the normal form. For large numbers scientific notation will be used on screen.

Floating point numbers, and in particular those with double precision, use more memory and are slower to calculate than long integers, long integers are slower and use more memory than integers. However the power and speed of HiSoft BASIC together with the large amount of available memory on the ST range means that these problems are much less noticeable than on other micros.

You will have seen by now that HiSoft BASIC does not force you to use any of these number format commands. As with many other commands, the language assumes a default setting of single precision if your exact requirements are not specified. There are also several associated commands that allow numbers of one type to be converted to another, for calculation or display.

It is possible to denote explicitly which type a number or numeric variable belongs to by the use of a certain suffix.

The % character denotes an integer.

The & character denotes a long integer.

The ! character denotes a single precision number.

The # character denotes a double precision number.

The suffix can be given as part of a number as in:

```
x = 234%
```

or as part of the variable name in which case that particular variable is unable to store data of the inappropriate type:

```
x! = 2.34
```

Constant literals, i.e. those that are not assigned to variables and which therefore will not change during the running of the program can again be integer, long integer, single or double precision floating point.

Of course any number or variable that looks like an integer can be forced to be stored internally as a floating point number by use of the appropriate suffix. A number or variable that has a fractional part can be forced to be an integer by assigning it to a variable with the % or & suffix and if this is done the number will be rounded to the nearest integer.

In the course of a calculation all numbers are converted to the same format as the highest precision number used anywhere in the current operation. The answer is also returned in a form that matches that precision. For example in the line:

```
x& = 100000& + i%*i%
```



`i%*i%` is evaluated as an integer and will overflow if `i%` is greater than 181. The result is added to `100000&` and this result is then turned into a long integer.

If an integer number, assigned to an untyped variable, undergoes a calculation such that it is converted to a number with a decimal fraction, i.e. the variable value is automatically converted to floating point. If the number is assigned to a typed variable the result of the calculation will be amended to fit that type.

There are also several associated commands that allow numbers of one type to be converted to another, for calculation or display, or which control the type of number a variable is capable of holding.

The following commands can be used to predefine what types of data a given variable name will store. They work by specifying the range of letters that are to begin the variable names for a given data type.

For example:

DEFDBL A-C, F

will mean that all variables that begin with the letters `A`, `B`, `C` and `F` will all be assigned double precision values even though in the actual statement that gives the value to the variable the number could look like single precision or even an integer, without any # sign.

DEFSNG range

will predefine a range of variable names to take single precision numbers,

DEFINT range

will define them to take integers and

DEFLNG range

to take long integers. You can also use

DEFSTR range

to define variables to hold strings; this allows you to dispense with the $ suffix for the relevant string variables although, of course, you must still include the $ for `MID$`, `LEFT$`, `INSTR$` etc.

All of the above definition commands are useful for freeing you from the need to explicitly define a variable type as it is being used, but they can be over-ridden by the use of the data suffixes described above. They are particularly valuable for catching input from the user that is required to be of a particular numeric type.

There are a suite of commands that change the internal format of a given numeric variable into another type.

`CSNG(x)`

converts `x` to a single precision number,

`CDBL(x)`

converts `x` to a double precision number,

`CINT(x)`

converts `x` to an integer and

`CLNG(x)`

converts `x` to a long integer. In both of the latter cases the fractional parts are rounded to the nearest integer. Try this:

```
x=22/7
PRINT CSNG(x), CDBL(x), CINT(x), CLNG(x)
```

`CLNG` is useful for promoting the result of a short integer calculation. As we said above, `100000& + i%*i%` will overflow if `i%` > 181 but `100000& + CLNG(i%)*i%` will allow a much wider range of values for `i%`.

The following commands are functions that act on a given variable, to return the integer part of that variable, although as you can see there are subtle differences in the way they work. These functions can be used to assign the integer part to a new variable, to print it on the screen or to allow it to be used in a calculation.

`FIX(x)`

returns the truncated integer part of the number, `x`, i.e. if `x` is 1.2, `FIX(x)` returns 1. Similarly if `x` is 22.99, `FIX(x)` will return 22 - it always rounds down with positive numbers. For negative numbers `FIX` still truncates the figures such that the overall effect is one of rounding up. If `x` is -22.99, `FIX(x)` will return -22.



`INT(x)`

returns the largest integer less than or equal to `x`. For positive numbers the effect is the same as `FIX(x)` but for negative numbers `INT` rounds down. If `x` is -22.99, `INT(x)` will return -23.

Finally there are two special arithmetic operators - `MOD` and `\` (backslash) which are used for integer division.

In an arithmetic expression such as `20.4\5.2` the backslash command stands for integer division such that both numbers are rounded to the nearest integers before the division takes place. The answer is also an integer value.

The keyword `MOD` returns the remainder of an integer division as an integer. `33.2 MOD 5` would return the value 3.

## Yet More Numbers - Bases

There is one more aspect of the way that numbers are stored and used that needs to be covered. In conversation we use exclusively decimal numbers, each column in a large number represents ten times the number to the right. This is known as *base ten arithmetic*. However there is no reason why we have to be restricted just to that system.

In their deepest darkest parts computers work exclusively in binary or base two arithmetic. Binary digits are either *on* or off and these two available options are used to build up larger and larger numbers. Each column in a large number represents twice the column to the right of it. In binary arithmetic 0 stands for 0, 1 for 1, 10 for 2, 11 for 3, 100 for 4, 101 for 5, 110 for 6, 111 for , 1000 for 8 etc.

However binary arithmetic soon gets unwieldy for people to use as the numbers increase in length rapidly, and are almost impossible to identify by just casual inspection. Programmers therefore tend to use the larger base eight (octal) arithmetic or base sixteen (hexadecimal) arithmetic. Because eight and sixteen are both powers of the number two, both octal and hexadecimal numbers have much more in common with the way that computers actually work than ordinary decimal numbers, and allow us to see patterns in the data that are meaningful to the computer.

Because these number bases are usually used for direct manipulation of the computer memory or similar, and would hardly ever be used for other calculations, we do not make provision to use fractional parts of numbers expressed in octal or hexadecimal - they are all expressed as the equivalents of decimal integers.

The command

```
OCT$(x)
```

will convert the (rounded) integer part of the number `x` to its octal equivalent. The result is stored as a string variable to avoid ambiguity, i.e. to ensure that calculations can not be attempted directly with octal numbers.

```
HEX$(x)
```

provides a similar conversion to hexadecimal. We have a problem with hexadecimal numbers that does not crop up in any base less than ten. We need a system for expressing numbers in the range 10-15 with just one figure. The convention used to do this is to use the letter A to stand for 10, B for 11, C for 12, D for 13, E for 14 and F for 15. So 8 = 8, B = 11, 10 = 16, 15 = 21, 1A = 26 and FF = 255.

Note for example that the total number of ASCII codes range from 0-255 and 255 is the binary number 11111111, and the hexadecimal number FF (both of which look much more significant than 255). This demonstrates how using other bases can be helpful in denoting meaningful numbers .

HiSoft BASIC allows the use of prefixes `&B`, `&H` and `&O` for specifying binary, hexadecimal and octal constants respectively - see the *Concepts* chapter for a full discussion of these different base constants.

Having broached the issue of how computers store and use numbers internally, it is probably worth pushing on with the subject as some terms will inevitably come up in later parts of the tutorial. The following brief explanation is not essential reading at this stage but will become increasingly important as your programs become more sophisticated.



## How Computer Numbers Work

Computers store all numbers and data internally in units known as *bytes*. A byte is an eight digit binary number that ranges from 00000000 to 11111111 which is 0 to 255 decimal or 0 to FF in hexadecimal.

Each digit in the binary representation of the number is known as a *bit*, this is short for Binary digIT. There are therefore eight bits to the byte.

However as microcomputers have become more sophisticated and powerful this relatively small size of data has become something of a bottleneck to performance. Routines that needed to work on large numbers for example had to retrieve the data in small 'byte sized' pieces, reconstruct the number in question, act on that data, break down the result into small pieces again and send it back into the memory. Most computer languages perform all of these tasks invisibly and the user knows little or nothing about it but there was an inevitable penalty in speed.

For more rapid movement and manipulation of data there has developed a need for larger data 'packets'. As well as the byte we now have the larger two byte equivalent, the sixteen bit word and the four byte equivalent, the thirty-two bit long word.

Within HiSoft BASIC, integers are held in 16 bits and long integers in 32 bits. These give the numbers -32768 up to 32767 for integers and -2147483648 up to 2147483647 for long integers. The reason that integers do not give 0 to 65535 is that it is generally more useful to use signed numbers rather than unsigned although some languages do give you the ability to choose between signed and unsigned integers.

Using signed integer arithmetic means that, in any integer, if the most significant (the leftmost bit) is set to 1 then the number is negative.

Remember, it is perfectly possible to program in HiSoft BASIC without concerning your self with any of these details. It is only when starting to manipulate the memory of the computer directly that they will become important.

## Using Logical Operators in Arithmetic

We have already seen how logical operators function to enable tests to be made between two different situations. The way these actually work internally is based on comparisons between the electronic patterns of two binary numbers - *bitwise comparisons*. In the case of a logical test the numbers that are usually compared are -1 (or another non-zero number) and O standing for true and false. Depending on the particular type of logical test that is made, the results of the comparisons will themselves resolve to a figure that will signify either true or false and the flow of the program will be altered in response.

You can also use these logical tests in expressions; the two operands will be converted to integers (or long integers) and then into binary and compared, bit by bit, according to which logical operator you are using, to obtain the result.

The following table shows the outcome of the comparison of each bit for the different logical operators:

| | 1 and 1 | 0 and 0 | 1 and 0 | 0 and 1 |
|---|---|---|---|---|
| NOT | 0 | 1 | 0 | 1 |
| AND | 1 | 0 | 0 | 0 |
| OR | 1 | 0 | 1 | 1 |
| XOR | 0 | 0 | 1 | 1 |
| IMP | 1 | 1 | 0 | 1 |
| EQV | 1 | 1 | 0 | 0 |

The type of comparisons that are made need not be confined to just true and false however, they can be used in just the same way to compare and modify two different binary numbers such as `10010011` and `11011110`. They can therefore be used in the following way, for example:

```
PRINT 162 AND 48
```

will be calculated by bitwise-`AND`ing the two binary representations of these integers as is shown on the next page:



```
162 AND 48 is

00000000 10100010  AND
00000000 00110000
-----------------
00000000 00100000  = 32
```

Note that we show, and calculate with, the full width (16 bits) of the binary representation of the integer - this is particularly important when using the operators `NOT`, `EQV` and `IMP` which may change a 0 bit to a 1 bit and thus often turn a positive number into a negative number.

See if you can predict the outcome of this program before you run it (preferably not in low-resolution for a reasonable screen width):

```
SUB L_Table(VAL x,y)
REM Set tab width to 9
WIDTH 80,9
PRINT "X","Y", "X AND Y", "X OR Y", "X XOR Y",
PRINT "X EQV Y", "X IMP Y", "NOT X"
PRINT
PRINT x, y, x AND y , x OR y, x XOR y,
PRINT x EQV y, x IMP y, NOT x
PRINT
END SUB

j=33
FOR i=4 TO 7
CALL L_Table (i,j)
NEXT i
```

These logical transformations are not of obvious use to the novice programmer but are extremely important for use in graphics work. `XOR` in particular has a useful effect that fairly easy to demonstrate.

Consider this:

`162 XOR 48` gives:

```
00000000 10100010  XOR
00000000 00110000
00000000 10010010  =  146
```

Now what about 146 XOR 48?

```
00000000 10010010  XOR
00000000 00110000
00000000 10100010  =  162
```

So, XORing a number with a constant and then XORing the result with the same constant gives us the original number. Because of this feature, XOR is used a in computer graphics to produce 'non-destructive' animation i.e. to allow pictures to move around on a screen without permanently obliterating what was in the background. First a certain graphics pattern is XORed onto the screen memory and hence displayed. XORing the same pattern a second time restores exactly the image that was there previously. The pattern is then repositioned slightly and XORed again to give the impression of movement across the screen.

The ability to XOR graphics images is implemented in the PUT statement, as shown in the following example.

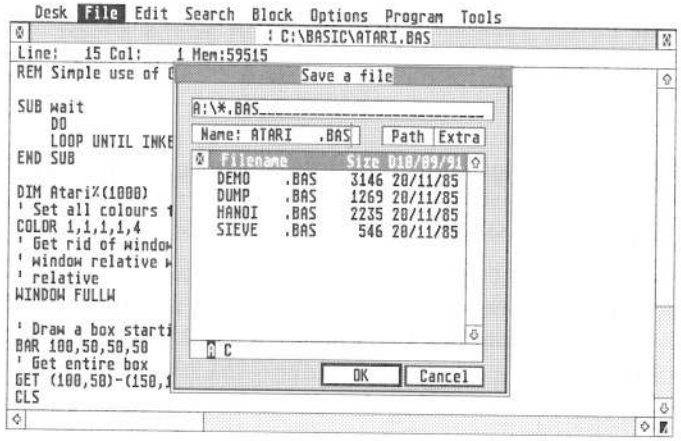
```
REM Simple use of GET and PUT

SUB wait
   DO
   LOOP UNTIL INKEY$<>""
END SUB

DIM Atari%(1000)
' Set all colours to black and fill to Atari
COLOR 1,1,1,1,4
' Get rid of window gadgetry because
' BAR is window relative while GET
' is screen relative
WINDOW FULLW

' Draw a box starting at (100,50)
BAR 100,50,50,50
' Get entire box
GET (100,50)-(150,150),Atari%
CLS
FOR i=0 to 100 STEP 5
   PUT (i,i),Atari%,XOR
   wait
   PUT (i,i),Atari%,XOR
   wait
NEXT i
```



Type in this program and run it. Now repeatedly hit any key, as quickly as you like, to watch a box of Atari logos appear and disappear while moving across the screen.

You might like to save this program. Make sure you have a disk in drive A with enough space on it (or room on your hard disk) and then select Save as from the File menu.

A file selector will appear (this is the Harlekin 2 file selector, yours may be slightly different):




*Saving a program*

Type in the name you want to call your program, using up to 8 characters for the filename, then a full-stop and then up to 3 characters for the file extension (say ATARI.BAS). Now click on OK or hit Return.

## Strings and Things

Strings, or text variables, differ from numbers in that, at least as far as the computer is concerned, there is no logical relationship between one character and the next - they are just a collection of letters and numbers 'strung' together.

Unlike decimal numbers where each character position signifies a tenfold relationship with the one to the right, strings cannot be used in calculations. This is true even when they look like numbers.

At times this distinction is very valuable - in calculating programs such as spreadsheets for example you can ask for a total of all numbers in a large block of data without worrying that the computer will also include bank account or telephone numbers as well.

However there are certain manipulations we can do with strings that are impossible with numbers. Long strings can be built up from shorter ones by a process called *concatenation*, which really just means adding them together - try this:

```
A$="Hello"
B$="John"
C$=A$+" "+B$
PRINT C$
```

This can be very useful for creating complex display layouts for example by making strings of tab characters - `CHR$(9)` - and by using carriage return codes - `CHR$(13)`.

Dividing strings into smaller portions is known as *string slicing*. The following functions all return strings that are at most equal to, or in in some way smaller than, the original that they were given as an argument.

```
LEFT$(string$,x)
```

will return the first `x` characters counting from the left of the string.

```
RIGHT$(string$,x)
```

will return the first `x` characters from the right of the string.

```
MID$(string$,a,x)
```

returns a string of `x` characters long starting from the position `a` in the original (the required length, `x`, can be omitted in which case the entire remainder of the string is returned).

`MID$` can also be used to alter the data held within a string by specifying a portion that is to be replaced with a second string using the syntax:

```
MID$(string$,a,x)=string2$
```

This will replace `x` characters from `string`, starting at position `a`, with the first `x` characters from `string2`.



```
INSTR(a,string$,string2$)
```

will search for the first occurrence of `string2` within `string` starting optionally from position `a`. If the second string is found the function returns the position of the first character of that string. Try this:

```
SUB Search (VAL A$)
IF INSTR(1,A$,"HiSoft")<>0 THEN
   PRINT "That's what we like to see!"
ELSE
   PRINT "Where's the HiSoft?"
END IF
END SUB

DO
   INPUT "Gimme some words, please";Test$
   Search Test$
LOOP UNTIL Test$=="END"
```

These commands can be used in conjunction with each other to build up more complex string manipulation expressions. They work well in conjunction with the function:

```
LEN(string$)
```

that returns the length of the string in question.

`LSET` and `RSET` can be used to place a string of text within a string variable of greater length such that the smaller string is assigned flush with either the left hand end (left-justified) or the right hand end (right-justified) of the larger. For these commands to work you must first create a larger string of the required size and the easiest way to do this is to use either the `SPACE$` or the `STRING$` commands. `LSET` and `RSET` are normally used when using files and `FIELD`ed variables but can be used to left- and right-justify any string.

```
A$=STRING$(30," ")
B$="HELLO!"
PRINT A$
PRINT B$
RSET A$=B$
PRINT A$
LSET A$=B$
PRINT A$
```

To illustrate the use of some of the above functions here is one that can be used as a complement to RSET and LSET to centre one text string within another.

```
FUNCTION centre$(main$,insert$)
STATIC a,b

   a = LEN(main$)
   b = LEN(insert$)
   IF b >= a THEN
      FNcentre$ = LEFT$(insert$ ,a) : EXIT DEF
   END IF
   c = (a-b)\2          ' integer division
   MID$(main$, c+1) = insert$
   centre$ = main$
END DEF

DO
   INPUT "A word";a$
   INPUT "Another word";b$
   PRINT centre$(a$,b$)
LOOP UNTIL a$=="END"
```

UCASE$ and LCASE$ convert all letters within a given string into upper case or lower case respectively. These are useful routines for tidying up the response given by a user:

```
DO
   INPUT "What is your name"; A$
LOOP UNTIL A$<>""
B$ = UCASE$(A$)
PRINT "HELLO "; B$
```

Of course punctuation and numbers within the string remain unchanged.

## Numbers to Text and Back Again

We have already touched on the subject of how letters are actually stored internally in the computer as numbers. These are known as ASCII (American Standard Code for Information Interchange) numbers and every letter or character that you can type with your ST/TT will have its own internal ASCII number.



Obviously it is vitally important that the computer knows at any one time whether the numbers it has stored in its memory are actually to be regarded as numbers or whether they really represent text.

Say for example that you have stored on a file somewhere your telephone number 0525718181. It will be essential that your programs are aware that this is actually a text representation of a number rather than a true number. It cannot be multiplied by any other figure, divided, assigned to a numeric variable, converted to double precision and so on, and in particular it should never be used to calculate your taxable income by your home accounts program!

It is to avoid such ambiguity that computer languages such as HiSoft BASIC insist that text variables are denoted by the suffix $ and the text strings themselves are enclosed in quotes.

However there are situations where it is important that the programmer has control over the way that the internal numbers are handled. It is possible to convert numeric variables to text, and vice versa, and we shall see that there are certain advantages to doing so.

### Text as numbers - the ASCII codes

Although text is always stored in memory as ASCII numbers the computer keeps track of which variables really represent letters and need to be converted to such before displaying them on the screen. There are however circumstances where it is valuable for the programmer to be able to access these numbers directly and decide what to do with them.

A typical situation may be where text data is stored in an unstructured way in memory, which could happen if it has been imported from another computer through one of the expansion ports. The actual data read and stored will be the ASCII number equivalents of that text.

At a later stage if you wish to display the information on screen you will have to explicitly signal to the computer that you want the data to be interpreted as letters rather than as numbers - unlike defined variables the computer will be unable to guess what each piece of data is to represent.

The keyword we use to convert a stored ASCII number to its text equivalent is CHR$(). Within the brackets we should put either a number or a numeric variable name. The computer will then convert this number to text using its internal ASCII table. *Appendix D* in the *Technical Reference* manual summarises the ASCII values for you.

The following short example will illustrate the way that the conversion of ASCII numbers to text works:

```
a = 72
b = 69
c = 76
PRINT CHR$(a);CHR$(b);CHR$(c);CHR$(c);CHR$(79)
```

which will produce the message HELLO.

It may seem a cumbersome way of doing things, but storing text in this way can be extremely useful. In particular it is valuable when you want to print or display characters that cannot actually be typed at the keyboard.

As well as producing some of the more unusual characters, different ASCII numbers can be used by your programs to trigger special effects.

Data files created by word processors, text editors and the like will contain ASCII numbers. Similarly all information exchange from the keyboard to the computer, from the computer to the printer, from the computer to the disk drive and from the computer to another computer will be as ASCII information. However what happens to that information when it arrives depends on the program that is running and on the computer's operating system.

For example, when ASCII codes are sent to the printer they do not all appear on paper. Some are regarded as control codes that instead trigger some sort of special effect. One such special effect code is ASCII number seven which produces a beep, either from the computer or the printer depending on where the code has been sent.

Another very important control code number is ASCII 27 which is known as the *escape* code. Most printers use this to signal the start of some command sequence that will produce a different type style such as bold or enlarged print or some other effect.



Many software programs will use their own routines to catch certain ASCII codes coming from the keyboard and to use them to trigger actions. One might do a simple test on which letter has been sent, but again ASCII codes are usually more flexible as it is possible to select unusual numbers that can only be 'typed' by some combination of letters such as Ctrl-X which in the ST/TT keyboard produces the character number 24. Alternatively an unusual code or sequence of codes can be assigned to one of the ten function keys.

Two of the most common and important ASCII control codes you will ever use are number 10 and number 13 which are the codes for line feed and carriage return, respectively. These are the codes that are sent whenever you press the Return key on your keyboard and signal the end of one line and the start of the next when you are typing e.g. using your HiSoft BASIC editor.

These codes are also extremely useful for controlling the display of information on the screen. Say you want to define the text variable ADDRESS$ to produce this output:

```
John Smith
22 Long Lane
Newtown
```

You can achieve this by using, all on one line:

```
ADDRESS$="John Smith"+CHR$(13)+CHR$(10)+"22 Long
Lane"+CHRS(13)+CHR$(10)+"Newtown"
```

There is an associated keyword which will return the ASCII number for any given letter or text variable. This command is ASC(). Example:

```
PRINT ASC("A") : PRINT ASC("*")
```

Sometimes the keyword ASC() is not appropriate because it always converts the characters to their ASCII table equivalent rather than to what we would regard as their direct numeric equivalent.

For example the text character **9** has the ASCII value of 57. The following line would therefore give the answer of 570.

```
x = ASC("9") : PRINT x * 10
```

What we need in this case is VAL. VAL searches a string expression for anything that can be interpreted as a number, integer or floating-point.

Try these out:

```
x = VAL("9") : PRINT x * 10
PRINT VAL("180 High Street North")
PRINT VAL("    -256+40=-216")
a$="26.04 - the price plus VAT")
PRINT VAL(a$)
```

VAL stops looking for a number as soon as it encounters something (apart from a space, tab or end-of-line) that it thinks cannot start a number.

### Numbers as text

Converting numbers to characters is essential whenever they are to be incorporated into a long string of text, or imported into a text data file such as one produced by a word processor etc. There are however other possible advantages in making the conversion.

We have already seen how text strings can be sliced into smaller pieces, or joined together, and in several ways manipulated in a very different manner to numeric variables. There are often occasions where it is useful to be able to do the very same things to numbers.

We have already seen above how to take a string and extract a number from it using VAL; the converse of this is where you want to convert a number into a text string - for this you use STR$. STR$ inserts a leading space or a minus sign, if the number is non-zero.

```
a=1066 : b$=STR$(a)+" and all that!"
PRINT b$
big=-98765.43 : b$=STR$(big)
PRINT LEFT$(b$,3);",";RIGHT$(b$,LEN(b$)-3)
```



## More Ways to Store Variables and Data

Computers are used all around us today, in almost every profession, business or service you can name, to store and manipulate large amounts of data. In many ways the use of variables lies at the heart of this power. For example a simple arithmetic expression can be placed inside of a logical loop and within a couple of hours it could have done literally thousands upon thousands of calculations if it was given a constant supply of numbers to work from. But where can this data come from?

We obviously need a way to store and read data that is less longwinded than the simple, hard-coded system we have employed so far of `variablename = data` or every business program would need thousands of lines that did nothing but assign values to variables (if you could think of enough names for them).

We have already mentioned disk files and these will often be the most common way of storing large amounts of variable data on your Atari ST/TT. However, for storing reasonably small amounts of data that is to be constant over the life of your program the DATA, READ and RESTORE commands are useful.

The keyword DATA is used to supply a list of several items of data, each separated from the next by commas. There can be many such DATA lines within each program. The keyword READ is used to look up this data, item by item, and assign it to a named variable, or several variables in turn (with the usual proviso that string data should not be assigned to a numeric variable etc.). When reading from the list HiSoft BASIC starts at the very first DATA line it finds. When finished it remembers its exact position in the list, ready to begin again following the next READ command.

The 'pointer' that the computer uses to keep track of its position within the list can be moved with the use of the RESTORE command. If used on its own RESTORE resets this pointer to the front of the first DATA line, but RESTORE can also be used with a specified line number or line label to move the pointer more selectively.

Text string data within the list does not have to be enclosed within quotes but you will get an error if you try to read such data into a numeric variable.

Try this little program:

```
DATA "A Box" ' quotes because of space character
DATA 100,50,100,150,200,150,200,50
RESTORE
READ a$
LOCATE 4,18
PRINT a$
x0=200 : y0=50
FOR a=1 to 4
   READ x1,y1
   LINEF x0,y0,x1,y1
   x0=x1 : y0=y1
NEXT a
```

Another system for storing large amounts of information is the use of dimensioned variables. These allow information to be held in a much more dynamic way than is possible with DATA statements, but make the organisation and management of the data much easier than with simple variables.

The keyword used to produce a dimensioned variable is DIM.

A dimensioned variable allows several items of information to be accessed under just one variable name by specifying a numeric position for the data within a list. Take this example:

```
DIM x(12)
```

This dimensions the variable x to hold 13 entries of data each accessed in turn as x(0), x(1), x(2), x(3), x(4), x(5), x(6), x(7), x(8), x(9), x(10), x(11) and x(12). This list of of variables, all stored under the same name, is known as an *array*. Data is placed in each of the entries of a dimensioned variable in just the same way as with a normal variable e.g.

```
DIM A$(21)
a$(12) = "Fred Smith".
```



It is also possible to define a two dimensional array as in DIM x(9,9) which will store 10*10=100 items of integer data. The data in a two dimensional array can be accessed as if part of a table, with each data item read by specifying the 'row' and 'column' number. Similarly a three dimensioned array can be made which can be looked on as perhaps many different tables held on several pages of a book. Three dimensions is the limit to what we can visualise in terms of everyday objects but the number of dimensions you can have in an array is unlimited.

Arrays of this form can be used to store data in a highly structured way. For example a series of string variables used by an address book program can be subdivided to store names, addresses, home telephone numbers, work address, work telephone number etc. and chosen parts of the data can be called up by specifying the appropriate array position.

It is not always necessary to use the DIM command when using dimensioned variables. Any variable name that is given an array position when it is first used will automatically trigger the formation of a dimensioned array of that name as in:

```
X(2)=20
PRINT X(1)
```

However when this system is used it creates a default array size of just 11 elements in each dimension referred to in the variable name.

> HiSoft BASIC provides a range of error checks on arrays which are controlled from the Compile... box. With Array checks on you will be told at runtime if you access an array element beyond the dimensions of the array. When you are sure that your program is safe from such errors you may want to turn Array checks off to increase the speed of the program and decrease its size. However, be warned; if you are relying on the auto-dimensioning of small arrays described above, it is a good idea to turn Array check warnings on before you declare your code finished and turn checks off. This will tell you if you have accidentally used an array without dimensioning it first. If you run a program which uses an un-DIMed array with Array checks off, a program crash may result.

The SHARED keyword can be used in conjunction with DIM to allow the array data to be shared between the main program and sub-programs without the need for a SHARED statement in each sub-program. Thus

```
DIM SHARED A$(20)
```

in the main program will allow the 21-element array A$ to be accessed by all sub-programs.

Many people find the dimensioning system confusing in that an array of say x(10) actually stores eleven items of data. This is because it includes x(0) as a legal data position. The command OPTION BASE 1 forces the lowest entry in an array to be position 1, in which case DIM x(10) would only allow ten items to be stored. You can reset the default situation with the command OPTION BASE 0 but there are no other options allowed.

REDIM is a command that allows the size of the dimensions of an array to be re-allocated. For example DIM x(9,19) can be REDIMensioned to the new layout of x(9,39) by:

```
REDIM x(9,39)     ' all data in x() is lost
```

Use this command with care as the data that is held in the array will be lost.

REDIM PRESERVE allows one dimensional arrays to be enlarged or truncated whilst retaining the data in the un-altered array elements.

When an array no longer is to be used it can be ERASEd, which clears all entries and reclaims the memory allocated.

When writing sub-programs the situation often arises where the programmer can anticipate that the sub-program will be passed a variable array as a parameter, but cannot anticipate the likely size of this array. Similarly you may wish to write program routines that can work on an array that may constantly be REDIMensioned. HiSoft BASIC has two functions which allow the size of a given array to be tested.

```
UBOUND(name,x)
```

returns the highest allowable entry of the 'xth' dimension of the the array allocated to the variable name.



```
LBOUND(name,x)
```

will do the same for the lowest allowable entry number, which will be either 0 or 1 depending on the setting of the OPTION BASE command before the array was first dimensioned.

Finally there is a command that can be used with any pair of variables but is particularly useful for manipulating and moving around the data held in variable arrays.

```
SWAP var1,var2
```

will exchange the values held in the two variables, providing they are both numeric variables of the same type, or both string variables.

Below is a complete example using these new array handling keywords:

```
DEFINT a-z

SUB Shell_Sort(Words$(1))
STATIC HowFAR, Top, i, j
   Top=UBOUND(Words$)
   HowFar=Top\2
   DO WHILE HowFar>0
      FOR i=HowFar TO Top-1
         j=i-HowFar+1
         FOR j=(i-HowFar+1) TO 1 STEP -HowFar
            IF Words$(j)<=Words$(j+HowFar) THEN
               EXIT FOR
            END IF
            SWAP Words$(j), Words$(j+HowFar)
         NEXT j
      NEXT i
      HowFar=HowFar\2
   LOOP
END SUB
```

Type this shell sort program into HiSoft BASIC and then devise a means of filling an array with some words and calling Shell_Sort to sort them into alphabetical order.

## Closing Down

STOP, END and SYSTEM are all commands that produce the effect of stopping the current program, closing all files and returning control to the calling operating system. They can be used as options within an IF...THEN branch or similar for bringing an end to the program. If the current program that is being executed ever 'runs out of lines' the effect is similar to placing an END command at the final line.

SYSTEM differs from END and STOP in that it suppresses the press any key message that normally appears when a HiSoft BASIC program finishes.

## The Screen

Producing output on the screen is at once one of the simplest and one of the most complex tasks possible. When you type PRINT "Hello" this single command triggers an enormously complex series of routines that produce exactly the correct pattern of dots on the screen, in exactly the correct place amongst the thousands of dots that there are, to create the image of the word Hello.

Normally all of these processes are controlled automatically by the operating system of your computer. This makes things very much simpler for you, but also inevitably limits the options available for tailoring the display.

On the other hand the ST/TT has some of the most powerful and flexible graphics abilities yet seen on a microcomputer. To exploit these, HiSoft BASIC allows a great deal of control over various aspects of the ST's operating system, including GEM, the *Graphics Environment Manager*. These commands are listed and explained in the *Supplied Libraries* chapter and the following section will only cover the commands briefly to put their usage into context.



## A Word About GEM

The Atari ST/TT range has an unusually sophisticated system of screen handling through the Digital Research GEM graphics environment. GEM allows the user to control the screen display via multiple windows, graphic icons and many other display features and options. These can in turn be linked to, and controlled by, the keyboard and mouse input devices.

Obviously, HiSoft BASIC has to be able to let the user access the power of GEM from within their own programs. However it would be completely impractical and unwieldly to provide special keywords for each of these functions, and their possible permutations.

HiSoft BASIC gives you a powerful and flexible interface with GEM by supplying a direct link to the GEM operating system routines via libraries and in addition, for the less adventurous, a complete high level GEM toolbox that hides most of the complexity of GEM, allowing you to build GEM programs quickly and easily.

We will briefly describe the use of the GEM libraries here and give an extended example in the next section. The use of the *HiSoft GEM Toolbox* is covered in the third part of this tutorial.

To make the process of using GEM directly from the ROM as easy as possible we have provided HiSoft BASIC libraries, each containing the routines that do this for you. Once the library is installed and accessible to your own programs you will be able to call any of these procedures by name, as if they were keywords, and pass the required parameters to them to get the result you want.

There are one or two pieces of bad news. Firstly, some of the GEM calls are detailed and fairly complicated and there is not sufficient room in this book to give more than the very briefest indication of what each one does. However, you may well find that studying the source code of the *HiSoft GEM Toolbox* will help you understand these low level routines.

The *Library Reference and HiSoft GEM Toolbox Reference* chapters in the *Technical Reference* manual are a reference to all the various functions and sub-programs provided, but if you intend to take GEM programming more seriously, you will probably want to buy a book that details the whole system at length and these, unfortunately, are few and far between. Some examples are listed in the bibliography.

Secondly the names used to refer to each of these controls (chosen by Digital Research and not by HiSoft) are hardly the sort of names that trip off the tongue or are easy to remember. If you find using the libraries difficult, you would be better off sticking with the *HiSoft GEM Toolbox*.

You may have discovered by now that your Atari also has a GEM-free 'text only' display mode called TOS. A HiSoft BASIC program will use this TOS mode if it does not contain any GEM functions or sub-programs and you have not forced your program to be a GEM program (see *Compiler Preferences* in the next chapter).

## Text on Screen

When placing text on the screen, which can be done in either TOS or GEM modes, there are really two broad classes of commands - those that position the text on the screen, and those that place the text at that position, and control the way that it looks.

The position where text will appear as soon as a `PRINT` or similar command is given is known as the *text cursor*. The screen of the computer is regarded as being divided into small squares, each big enough to hold one letter, of which there are eighty across and twenty down on a regular high or medium level screen.

The current horizontal position of this cursor can be determined by using the function `POS`, the vertical position by the function `CSRLIN`. The text cursor can be re-located at any of these text squares by the command `LOCATE x,y` where `x` is the vertical position and `y` the horizontal.



Either `x` or `y` can be omitted if their current values are to remain unchanged (although the commas that separate them must remain if `x` is omitted). One other optional parameter makes the cursor invisible or visible - see the *Command Reference*. Try:

```
LOCATE 10,10
PRINT "Jim"
LOCATE 9,10
PRINT "Hello"
```

Which will produce the output:

```
Hello
Jim
```

Cursor positioning can also be achieved by typing the two 'invisible characters' - spaces and tabs.

The keyword `SPC(x)` can be inserted in any `PRINT` statement and it will cause the cursor to skip `x` spaces before the specified text is displayed.

`TAB(x)` used in a similar situation will cause the text cursor to jump to column number `x` (if `x` is less than the current position, the cursor will move down by one screen line first). Note also the use of `STRING$` and `SPACE$` which can be used in `PRINT` statements.

```
PRINT SPC(10), "Hello"
PRINT TAB(10), "Hello"
```

will both produce the result:

```
          Hello
```

as will:

```
Let A$=SPACE$(10)
PRINT A$+"Hello"
```

Note that the command `TAB(x)` does not really print a tab character but causes the cursor to jump to the required spot on the screen. The true tab character is designated by `CHR$(9)` which can be used by many printers, and some programs such as word processors, to trigger a jump to preset tab positions. This is the character usually returned by the tab key on the keyboard.

With the cursor positioned where it is required, text can be displayed by the PRINT command. As we have seen even this has its own parameters that control the subsequent positioning of output - if text or a string is terminated by a semi-colon future output begins adjacent to that output. If it is terminated with a comma the next item to be printed starts at the next available x-column position on the screen. The default tab width is 14 and this can be changed using the WIDTH command.

If text reaches the end of a line it automatically wraps around to the start of the next. However it is possible to define a smaller maximum line size by the use of the WIDTH command.

```
WIDTH 20
PRINT "Hello Hello Hello Hello Hello"
```

will produce:

```
Hello Hello Hello He
llo Hello
```

Normally when the PRINT command is asked to display data, such as a double precision number, it shows it on screen in a way that reflects its internal format. Often, however, it is not desirable to have too many decimal places displayed, or we may wish to manipulate the display in some other way e.g. to add currency symbols to the front of the data. The PRINT USING command does all this and more and is covered in detail in the *Command Reference* section.

The WRITE keyword is also used to place data on the screen but it differs from PRINT in that it does not respond to any formatting commands.

Try this example:

```
REM An example of using WRITE
a=42 : b=a MOD 5 : c#=22/7
WRITE "The answer is "; a
WRITE b*5, c#
```

Run the program - is that what you expected?

Any strings that are printed will be enclosed in quotes.



In practice WRITE is more useful when debugging, or when producing disc files where the internal format of the number needs to be preserved. PRINT is of more value for the screen display in most other cases because of its extensive formatting control.

Finally, an important command for controlling the text, and indeed graphics, display is CLS which means 'clear screen'. This command will also return the cursor to the top left hand edge of the screen ready for printing new data.

### TOS vs GEM - when is text not text

The ST/TT comes supplied with a standard character set which gives the only characters available for printing text in TOS mode.

One of the main features of GEM, on the other hand, is that it is possible to use several different fonts and text styles, such as italic or bold print, on screen. These special font styles cannot be handled in the same way as normal text - they are in effect 'pictures of text' that are drawn on the screen. The ordinary HiSoft BASIC commands for displaying and manipulating text are not be capable of manipulating them in the required way, and instead your program will have to pass commands through the GEM interface library - see the *Supplied Libraries* chapter.

## Graphics

### Two ways to produce graphics

There is more than one way to produce a picture on your screen, but to appreciate the difference between the various techniques involves an understanding of how the screen display memory works.

The image you see on the screen is made up of a matrix of small dots known as *pixels* - in the high resolution ST monochrome mode there are 640 * 400 pixels which makes 256000 dots in total.

Other ST display modes have 640 * 200 pixels in 4 colours (medium resolution), which is 128000, and 320 * 200 pixels in 16 colours (low resolution) which makes 64000 pixels. The more pixels there are on the screen the higher the screen resolution. The TT has many new screen modes with differing resolutions.

As with any form of data, if the computer is to remember the image that is to be retained on the screen it needs to put aside a certain amount of memory for the task. Because of the different information that needs to be stored about the colours, the amount of memory it takes to store all of the information about a given dot increases as the screen resolution decreases. The net result cancels out so that the total screen display uses a constant amount of memory - 32000 bytes.

It is possible for sections of the screen to be accessed directly, altered, moved, copied or saved to disk. A disk file that contains the entire contents of the screen will be 32000 bytes large on the ST and a massive 153600 bytes under the new TT screen modes.

Say that we have a screen with a simple graphic image on it - one straight line. The program that draws that line may be only one command long and can be saved in a disk file that is less than 1K in size. Every time we wish to create that graphic image again the program can almost instantly be run.

However if we wish to save the entire memory image of that screen the file created will be 32 times as large, and will probably still be loading long after the first version has finished.

Alternatively a detailed graphics image, say a landscape scene, may require an complex program of many hundreds of lines to re-create it. If the screen picture is a digitised image of a video picture or a creation from a computer art package it may be practically impossible to write a program that could reproduce it.

In this case a simple file that stores the pixel data for the screen image will be the most memory efficient and speedy system to use.

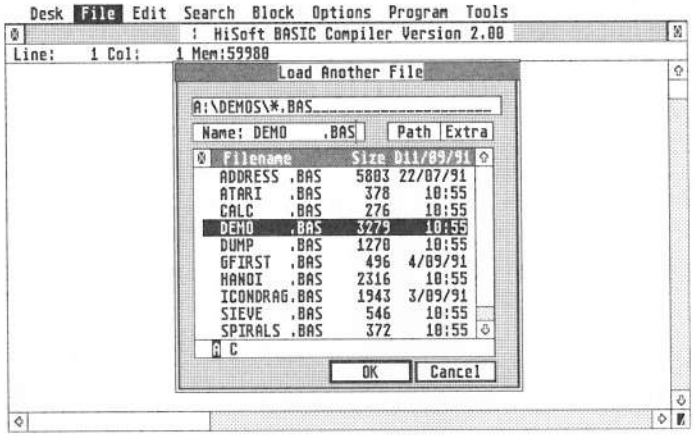
There are other advantages to using the latter system - you can see some in the demo file DEMO.BAS on your HiSoft BASIC master disk; it can be used to animate screen displays by rapidly moving part, or the whole, of the screen memory contents.

You can load this program in as follows:

- Insert the backup of your HiSoft BASIC disk 2 in drive A.



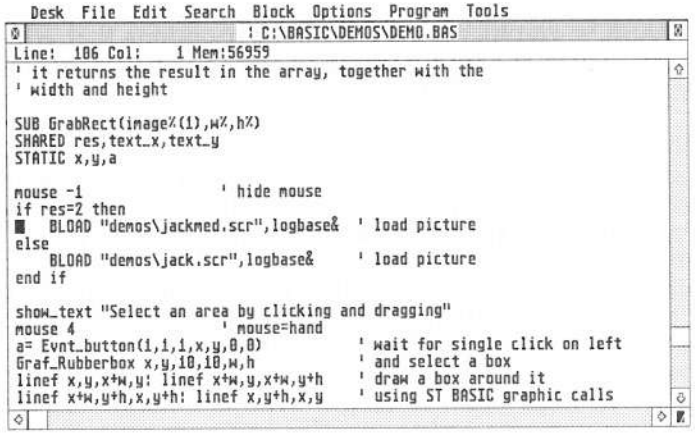
- Select Load... from the File menu and the file selector will appear (this is the Harlekin 2 file selector, yours may be slightly different):




*Loading DEMO.BAS*

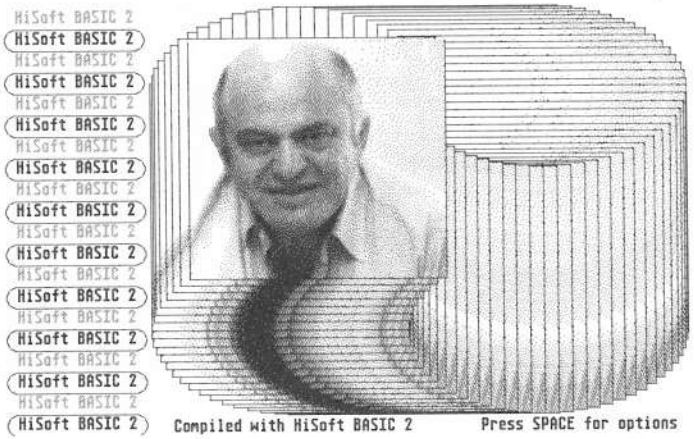
- double-click on DEMO.BAS on the file selector,
- type Alt-G and then enter 106 in the dialog box; you should see line 106 appear in the middle of the screen:

```
bload "\tutorial\jackmed.scr", logbase&
```




*Line 106 of* DEMO.BAS

This is the line that loads a graphic screen from the disk ready for manipulation. The `logbase&` is an XBIOS function call (detailed in *The XBIOS Library* section in the *Supplied Libraries* chapter) that returns the address of the start of the screen. It's also quite fun to run this program ...




DEMO.BAS *running in monochrome*

It is a good exercise to study the listing for this program to see how the program first tests to see which resolution mode the computer is operating in, and then BLOADs the appropriate disk file into the screen memory.

HiSoft BASIC's graphics keywords contain options that will make it easy for you to choose whichever of the above approaches best suit your needs, or to mix them at will.

As with text the following keywords require a means of signalling to the computer exactly where you want the graphics to appear as well as what graphic image it is to print. The current position where graphics will appear is known as the *graphics cursor*.

The following commands all produce certain simple shapes on the screen at specified positions. Their full range of permutations and options are detailed in the *Command Reference* chapter.



Movements of the graphics cursor can be expressed in absolute coordinate terms like that used to move the text cursor (although of course the range of coordinates available changes depending on the current screen resolution). Alternatively it is sometimes possible to express the required movement relative to the current graphics cursor position using STEP.

LINEF is used to draw a line or a box on the screen in a chosen line style and colour. CIRCLE does the same for any size of arc or circle. The keyword ELLIPSE can be used to draw ellipses.

```
LINEF 100,100,200,100
```

will draw a line from the pixel position (100,100) to the pixel position (200,100).

BAR, PCIRCLE and PELLIPSE are used to draw shapes that are filled by a specified colour or tile pattern. The ability to rapidly and automatically fill shapes with a given pattern, which can be extremely complex, is one of the most powerful features of GEM and can give a very professional look to your programs if used with restraint.

PSET and PRESET both draw a pixel dot at the specified position on the screen in the specified colour. If the colour option is omitted PSET will default to the current foreground colour but PRESET will default to the background colour. POINT is a complementary command that will read the colour displayed at specific pixel position.

An example of the use of these graphics commands follows after the next section.

## Using Colours

The Atari ST computer is capable of displaying a maximum of sixteen colours at once. The current colours for printing and drawing can be selected by the programmer from a list of 0-15, detailed in the *Command Reference* section, under COLOR.

The extra palette (4096 colours) of the STE and TT can be accessed from HiSoft BASIC in two ways: via the built-in PALETTE statement and via the GEM VDI vs_color library function.

Using the PALETTE statement, the hex digits used to choose the rgb colour intensity can range from 0 to F (4 bits) and not just from 0 to 7 (3 bits) as on the original ST. But, to preserve compatibility with the ST, the top bit (bit 3) is used as the least significant bit, so that intensity 8 on the STE comes between intensity 0 and 1 on the ST and F is more intense than 7.

| STE | 0 | 8 | 1 | 9 | 2 | A | 3 | B | 4 | C | 5 | D | 6 | E | 7 | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Intensity of gun* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| ST | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | |

Consider the following program:

```
palette 2,&300
print "this is ST mid-intensity red"
palette 3,&B00
print "this is STE mid-intensity red"
```

If you run this on an STE, you can probably just see, the two reds are slightly different - on an ST they would be the same. A program using the VDI would look like:

```
library "gemvdi"
vs_color 2,1000,0,0
print "this is the brightest red"
vs_color 3,950,0,0
print "this is not quite as bright"
```

Here the two colours are the opposite way round than in the previous program. Using the VDI obviously requires the extra library call, but has the advantage of asking for the brightest possible colour without different code for the STE and the ST. Using the device-independent VDI also means that your programs should work with any add-on graphics card that may have support for an even wider range of colours.

The use of these colours is ultimately controlled by the type of monitor chosen by the user and the resolution of the display chosen by the programmer.



Owners of a monochrome monitor screen can only use the high resolution two colour (colours 0 and 1) display. On an ST colour screen there is a medium resolution mode which allows 4 colours (0-3) and a low resolution mode which allows all 16 colours to be displayed at once. Some add-on graphic cards support 24 bit colour which gives you over 16 million different possible colours whilst on the TT there is a TT low resolution mode that gives you up to 256 colours at once.

COLOR (with apologies for the American spelling) can be used to choose the colour with which text is printed and the colour of the background, or 'paper', on which it is displayed. When programming in TOS mode these are the only options available.

In GEM mode, the COLOR command can also be used to select the colours used for line drawing and the pattern used as a fill in, say, the PCIRCLE and PELLIPSE commands.

PALETTE allows the programmer to change the actual colours displayed on the screen by any of the sixteen colour numbers. In this way the displayed colours can be changed (on a colour monitor) even when, say, in the ST medium resolution mode which restricts the screen display to the use of numbers 0-3 only.

Try the following program on a colour display, in ST medium resolution:

```
DEFINT a-z

' Use the XBIOS library
LIBRARY "xbios"  ' Open library for the vsync command

' Draw three boxes in a triangular formation
FOR i=1 to 3
   x=270       ' Set up co-ordinates for each box
   y=25        ' depending on the screen resolution
   if i=2 THEN x=x-75: y=y+65
   if i=3 THEN x=x+75: y=y+65
   COLOR ,i          ' Select fill colour only
   BAR x,y,100,50    ' Draw the box on-screen
NEXT i
```

```
' Cycle through all the colours
FOR red=0 TO 7
   FOR green=0 TO 7
      FOR blue=0 TO 7
' Initialise each of the three colours we use
         PALETTE 1,red*&h100+green*&h10+blue
         PALETTE 2,red*&h10+green+blue*&h100
         PALETTE 3,red+green*&h100+blue*&h10
         vsync    ' Pause for up to 1/50th of a second
         vsync
      NEXT blue
   NEXT green
NEXT red

' Reset the colours to the standard ones
PALETTE 0,&h777
PALETTE 1,&h000
PALETTE 2,&h700
PALETTE 3,&h070
```

A colour that is altered by the PALETTE command changes instantly on the display without anything having to be redrawn. Even when in the low resolution sixteen colour mode, being able to alter the colour that is displayed by a given number is extremely valuable. Many forms of graphics effects and computer animation are possible as a result.

For instance two different colour numbers can be set by the PALETTE command to actually display the same result on the screen. An image drawn in one colour will therefore be invisible against a background using the other. A new PALETTE command that sets the image colour number to a different setting will make that image appear instantaneously.

## Example GEM Program

Here's a program that uses GEM (the VDI) without the *HiSoft GEM Toolbox*, which will be covered later in this chapter. This program should help you to understand the principles of using the GEM libraries.

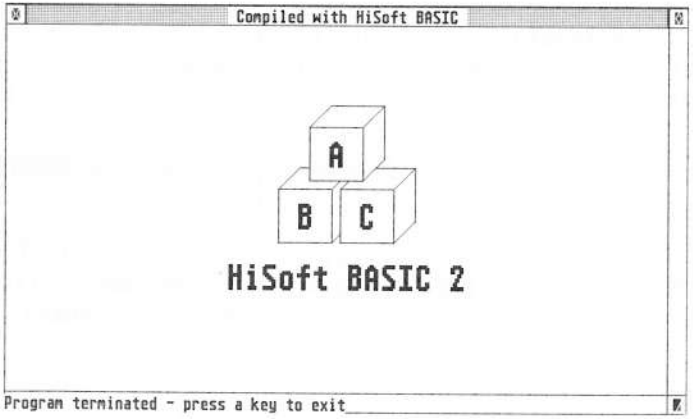


```
DEFINT a-z

REM Use the GEM VDI and XBIOS libraries
LIBRARY "gemvdi", "xbios"

CONST length=40,skew=15,xstart=260,ystart=110,ch_h=32
fb$="HiSoft BASIC 2"

DIM ch(7)

REM Sub-program to draw a 3-D box with a letter
SUB draw_box(BYVAL x,BYVAL y,BYVAL ch$)
LOCAL pts(11) : SHARED ch_x, ch_y

REM Draw outline of box
   pts(0)=x          : pts(1)=y
   pts(2)=x+skew     : pts(3)=y-skew
   pts(4)=x+length+skew    : pts(5)=y-skew
   pts(6)=x+length+skew    : pts(7)=y+length-skew
   pts(8)=x+length   : pts(9)=y+length
   pts(10)=x         : pts(11)=y+length
   v_fillarea 6,pts()        'main filled polygon

REM Now draw 3 lines to complete 3-D box

   pts(2)=x+length        : pts(3)=y
   pts(4)=x+length+skew   : pts(5)=y-skew
   v_pline 3,pts()          '2 lines

   pts(0)=x+length        : pts(1)=y+length
   v_pline 2,pts()          'final line

REM Draw the 1-letter text, centred in the box
   v_gtext x+(length-ch_x)/2,y+(length+ch_y)/2,ch$

END SUB
```

```
REM The main program
vsf_color 1                         'set black border
vsf_interior 0                      'set hollow attribute
vst_height ch_h                     'height of characters

vqt_extent "A",ch()
ch_y=ch(7)-ch(1) : ch_x=ch(2)-ch(0)

draw_box xstart,ystart,"B"
draw_box xstart+7*length/6,ystart,"C"
draw_box xstart+7*length/12,ystart-7*length/6,"A"

fb_width=len(fb$)*ch_x : fb_height=ch_y
fb_x=xstart+(13*length/6+skew-fb_width)/2
fb_y=ystart+length*3*fb_height/2

REM Display picture title
v_gtext fb_x,fb_y,fb$
```

Daunting? Not really ... let's take it apart. First of all, type it in and run it in ST medium or ST high resolution; for simplicity it only works effectively in these resolutions - see the ABC.BAS program on Disk 2 for a resolution-independent version.

Once it is running successfully it should produce the following output:




*GEM is as simple as A B C!*



Now let's see how it's done ...

First of all, what are we trying to draw? Answer: three 3-dimensional boxes, with letters inside them and a title below all this. How are we going to do the boxes?

Look at the following diagram:

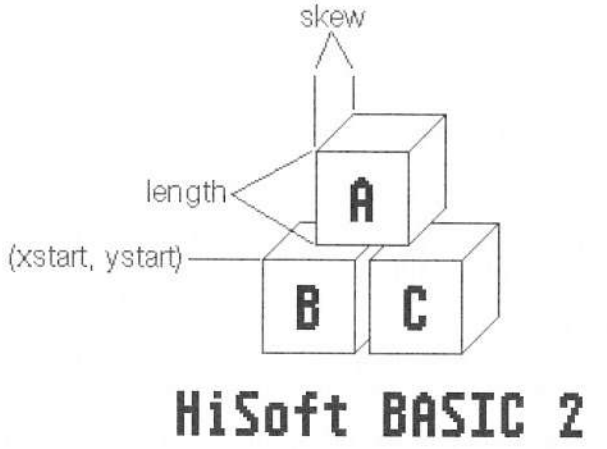



*Figure 3.1 The problem ...*

We have to draw 3 boxes, all the same (except for the letter inside it), so it makes sense to have a sub-program to draw one box. To make a box, we could just draw 9 lines but there is a GEM call to help us out ...

The GEM VDI sub-program `v_fillarea` is documented in *Chapter 5* under *GEM VDI : Drawing Primitives*. From the description we see that `v_fillarea n,pts()` fills a polygon of `n` points which should be given in the array `pts()`. Look at the box in *Figure 3.1*, formed by points 1, 2, 3, 4, 5, and 6 - this is a closed polygon and we could therefore draw it with `v_fillarea`.

What are the points? Well, if point 1 is `(x,y)` and the length of each side of the box is `length` and the depth, measured vertically and horizontally, is `skew` then the points are as follows:

Point 1 : (x, y)
Point 2 : (x+skew, y-skew)
Point 3 : (x+length+skew, y-skew)

Point 4 : (x+length+skew, y+length-skew)

Point 5 : (x+length, y+length)

Point 6 : (x, y+length)

So, to draw this polygon we must set up 12 (2*6) elements of the `pts()` array to contain the `(x,y)` co-ordinates of these 6 points:

```
REM Draw outline of box
  pts(0)=x      : pts(1)=y
  pts(2)=x+skew   : pts(3)=y-skew
  pts(4)=x+length+skew : pts(5)=y-skew
  pts(6)=x+length+skew : pts(7)=y+length-skew
  pts(8)=x+length : pts(9)=y+length
  pts(10)=x     : pts(11)=y+length
```

To draw this polygon we should first set up the fill pattern and the colour index of the fill; this is done with `vsf_interior` and `vsf_color` respectively - see the *Reference Manual : Chapter 2 GEM VDI : Attributes*. There is some logic to these GEM names: the `v` stands for *VDI* (Visual Display Interface), the `s` for *set* and the `f` for *fill* - these are *VDI set fill commands*.

So, to draw this polygon we set up the `pts()` array as above and then do:

```
  vsf_color 1      ' black colour
  vsf_interior 0   ' hollow fill. Try 4 for fun!
  v_fillarea 6,pts()      ' draw the polygon
```

Now, drawing the polygon hasn't given us our 3D box yet - we need to add lines from point 7 to point 1, from point 7 to point 3 and from point 7 to point 5 (see *Figure 2.1*). We need another VDI call to do this (we could use the BASIC `LINEF` command but with difficulty since `LINEF`'s graphic origin is the top left of the *window* whilst GEM's graphic origin is the top left of the *screen*!). Scout through the *Reference Manual : Chapter 2 GEM VDI : Drawing Primitives* and you come across `v_pline`, which would seem to do the job.

Point 7 has co-ordinates `(x+length,y)` and we should be able to draw the 3 lines in two calls to `v_pline` like this:



```
REM pts(0) and pts(1) already set to point 1
  pts(2)=x+length      : pts(3)=y      ' point 7
  pts(4)=x+length+skew : pts(5)=y-skew ' point 3
REM Join point 1 to point 7 to point 3
  v_pline 3,pts()    '2 lines (3 points)


REM pts(2) and pts(3) already set to point 7
  pts(0)=x+length      : pts(1)=y+length ' point 5
  v_pline 2,pts()    'final line (5 to 7)
```

`v_pline` draws lines with the current line attributes (routines begin with `vsl` - *VDI set line* i.e. `vsl_type` etc.); we don't need to change the default styles for this program.

So, we've drawn a 3D box; now we have to put a character in it. Where shall we look to find a routine to draw text on the screen in GEM? Let's look under *GEM VDI : Drawing Primitives* in the *Reference Manual : Chapter 2*; there is `v_gtext x,y,text$` which writes the string `text$` on the screen at position `(x,y)`. How about choosing the size of text? The command says that the text *attributes* are used - time to look in *GEM VDI : Attributes* again. If you are beginning to catch on to GEM's naming conventions you might be able to work out that the routine will start with `vst_` (*VDI set text*) and is called `vst_height`.

When using the character set in the Atari ROMs (as opposed to using GDOS), the largest character that GEM can draw is 32 pixels - we'll have a `CONST`ant in our program that represents this character height: `ch_height` (you can try changing it) and then we can use `vst_height ch_height` to set up 32-pixel high characters.

So we now need a bit of arithmetic to work out where to place this character so that it appears in the middle of our box. First we need to know the width and height of a character - can GEM help? The routine would be a *VDI* call and a *query* on *text*; `vqt_`. Look under *GEM VDI : Enquiries* in the *Reference Manual : Chapter 2*, there's a routine called `vqt_extent` that returns (into an array) the co-ordinates of a box needed to enclose a text string.

Having got these co-ordinates we can do a little subtraction to obtain the width and height of the character:

```
DIM ch(7)

vqt_extent "A",ch()
ch_y=ch(7)-ch(1) : ch_x=ch(2)-ch(0)
```

`ch_y` is the height of the character and `ch_x` is the width. To put the character in the centre of the front face of the box (the face is `length` wide by `length` high, remember), we need to put the bottom left of the character at co-ordinate (`x+length/2-ch_x/2, y+length/2+ch_y/2`) where (`x,y`) is the top left of the front face. Look at *Figure 3.1* to see that this is right. Keep looking!

So, this should do it:

```
v_gtext x+(length-ch_x)/2, y+(length+ch_y)/2, "A"
```

That effectively completes the `draw_box` sub-program. Notice that we have passed the top left (`x, y`) co-ordinates as parameters and also the letter that is to be contained in the box is a parameter (the third one).

The variables `ch_x` and `ch_y` (character width and height) which are calculated outside the sub-program are SHARED by it so that it can use them and the array of points that make up the cube (`pts()`) is declared as LOCAL to the sub-program.

> We could have declared the `pts()` array to be STATIC (which would seem more logical since you normally only use LOCAL variables when you have recursive procedures in which you want a new variable on each call) but there is a problem with declaring arrays as STATIC - the syntax of STATIC demands that you specify the number of *dimensions* (not elements) when using it e.g. `STATIC pts(1)`, a 1-dimensional array.



> Thus `pts()` must be DIM'd elsewhere (to tell the BASIC how many elements it has (if greater than 10)). However, it can't be DIM'd in the sub-program since you would get a `re-dimensioned array` error on the second time you called the array; therefore `pts()` must be DIM'd outside the sub-program and thus must be DIM SHARED to ensure that the sub-program can get at it. Or it could be REDIM'd within the sub-program without being declared outside. Either way is messy and therefore it is much better, for readability and neatness, to use LOCAL for arrays within sub-programs.
>
> It often, but not always, makes sense to declare arrays as LOCAL since, in general, arrays take up a lot of memory space which might be left around if the array was STATIC and you didn't clear it out after use; LOCAL arrays are automatically cleared on exit from the sub-program or function, thus releasing the memory.

Now we've completed the `draw_box` sub-program and all that remains is to call it 3 times (for 3 boxes) and then to write some text underneath the whole thing. Again, we'll use `v_gtext` to write the message and do a little arithmetic to make sure that the text message is centred under the boxes:

```
REM Draw 3 boxes with letters in them
draw_box xstart,ystart,"B"
draw_box xstart+7*length/6,ystart,"C"
draw_box xstart+7*length/12,ystart-7*length/6,"A"


REM Centre text below boxes
fb_width=len(fb$)*ch_x : fb_height=ch_y
fb_x=xstart+(13*length/6+skew-fb_width)/2
fb_y=ystart+length*3*fb_height/2

v_gtext fb_x, fb_y, fb$
```

Finally, we can spice up the message by using `vst_effects`; try putting `vst_effects 8` or `vst_effects 18` before `v_gtext fb_x, fb_y, fb$`.

We hope that has given you some insight into using GEM - it is really not as daunting as it first looks and well worth the initial effort!

Of course, if you find the GEM functions and sub-programs difficult to remember or clumsy to use you can always re-write and re-name them e.g.

```
REM Set the fill colour and fill style
SUB fillset (BYVAL fstyle, BYVAL fcolour)
   vsf_interior fstyle
   vsf_color fcolour
END SUB
```

Or you might like to use the *HiSoft GEM Toolbox*, which will be described in detail a little later.

## Menus, Dialog Boxes, Icons etc.

There will come a time when you may want to program all the trappings of a professional GEM program - drop down menus, alert boxes (no user interaction allowed), dialog boxes (allowing the user to make choices) and icons. These structures are normally trees of lower-level objects and they are usually held in a resource file that is separate from your program.

Programming these resources is possible with HiSoft BASIC by itself but you will find it far easier to use a *Resource Construction Set* (RCS) to help you. An RCS facilitates the design of menus *et al*, prepares the resource file for you and also gives you most of the constants needed to control these resources from within BASIC.

HiSoft WERCS is an example of a modern resource construction set and it is included with HiSoft BASIC 2. An extended example of using WERCS together with the *HiSoft GEM Toolbox* is provided later in this chapter.

## Blitting

As explained above there is an alternative approach to producing graphics effects which relies on using keywords which directly access and manipulate areas of the screen display memory. Moving or copying such areas of memory, or switching one area of screen memory with another, can be used to produce outstanding animation effects. Moving bits of screen memory around is known as *blitting* (the word stems from bit block transfer).



GET and PUT allow the program to pick up a specified rectangular block from the screen display and place it in a different position on the screen. An array of the appropriate size has to be specified for storing the data. A block that is picked up by GET is not automatically obliterated from the screen but merely copied into an array.

Several arrays can be filled using the GET command. These can also be copied to a sequential disk file in the same way that any array would be, ready for subsequent re-loading. In this way you could prepare a program that contained several digitised images, for example, that can be used for illustrations, or could be subsequently PUT into the same position to produce an animated effect.

In the past these techniques have been felt to consume too much memory to allow their frequent use in micro-computer programs, but most ST/TT machines have lots of RAM to spare and can hold many such arrays at once, leaving you free to produce incredibly sophisticated effects with ease.

The PUT command has several options that allow different graphics and animation effects to be achieved using the techniques described in *Using Logical Operators in Arithmetic* where an example of GET and PUT was also given. These options are detailed in the *Command Reference* section. Trial and error is also a good way of seeing how these routines work in practice - experiment with PUTting picture blocks onto a blank screen and onto a screen that already contains a detailed image to see how different effects can be achieved (people using a colour display will find a lot more interesting effects are possible than with a monochrome monitor).

Again the DEMO.BAS demonstration file gives an excellent example of the GET and PUT features in operation.

There is another technique that you could use to produce similar effects, on a screen by screen, basis which does not involve moving large blocks of memory around but instead changes the position of the pointer which tells the computer the address where, within the total amount of RAM, the screen display memory is to be found.

Within the memory you can hold several 32K areas of data, each of which has been created on the screen (using perhaps a drawing package or the commands available in HiSoft BASIC), and BSAVEd to disk but which have been BLOADed back into different positions. It is then possible to instruct the ST that each one of these start locations in turn is to be regarded as the address where the screen display memory begins.

You are in effect moving the screen display pointer rather than moving the data held in the memory itself. The result is that the image seen by the user changes much more rapidly and much more smoothly than it would have done otherwise, giving a more professional finish to your programs.

All of the HiSoft BASIC graphics commands will automatically take account of any such screen changes that you make.

> Many commercial games programs use these techniques to produce a high quality display. Two 'screens' are held in memory at once and the program always makes any movement or alteration to the display data by working on the screen that is not currently on show to the user.
>
> To make changes to the screen display pointer, and to check the current setting of that pointer (for BLOADing new images for example) you will have to make calls to the XBIOS and GEM Libraries (see physbase&, logbase& and setscreen in the *XBIOS Library* section of *The Supplied Libraries* chapter - and be careful!).

## Windows

Anyone who has had experience of using the Atari ST/TT will quickly learn what windows are, what they can do and how they can give your programs a professional and polished air. Many languages will only allow you to use windows by making rather complicated calls to the GEM operating system.

HiSoft BASIC however comes with a comprehensive selection of built-in commands that are designed to make the process of calling windows as easy and as fluent as possible. In addition, the *HiSoft GEM Toolbox* provides extensive high level routines to make window handling even easier.



The built-in commands are fully documented under the WINDOW keyword, in the *Command Reference* section and we will give an example here. For some help on using the windowing features of the *HiSoft GEM Toolbox*, see the examples later in this chapter.

WINDOW is a remarkably compact keyword requiring you only to provide several parameters such as the size, position, nature and contents of the window and HiSoft BASIC and GEM do all of the work of drawing the image on screen etc.

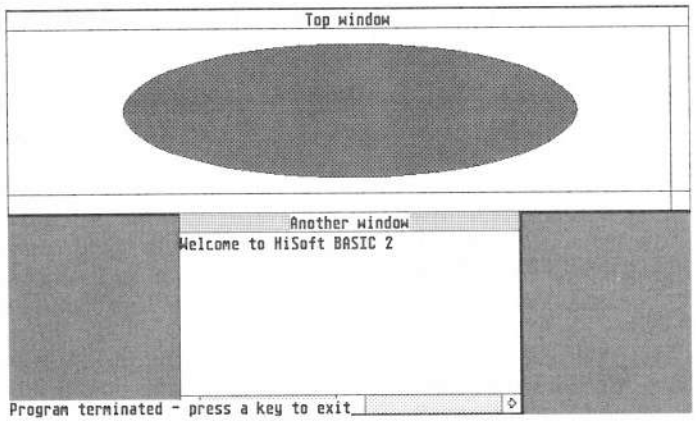
Note that when using multiple windows each is referred to by a specific identification number in a way that parallels those required for managing several disk files at once.

An example program that manipulates two windows is shown below:

```
DEFINT a-z
'
' Get the position, width and height of the free
' window that HiSoft BASIC gives you

WINDOW GET 2,1,wx,wy,ww,wh

' Re-locate and name the free window you get
' when running a HiSoft BASIC GEM program.
' This always has id number 2.

WINDOW LOCATE 2,wx,wy,ww,wh\2
WINDOW NAME 2," Top window "

' Set up a shaded (5,2) fill pattern for the ellipse
COLOR 1,1,1,5,2

' Draw an ellipse in window 2
PELLIPSE ww\2, wh\5, ww\3, wh\6

' Open another window: id number is 1, top left is
' below the other window, in the centre of the screen.
' The 1+512+1024+2048 means the window has:
' a title bar, left & right arrows and a
' horizontal slider.
' Type the next two lines in one line

WINDOW OPEN 1," Another window ", wx+ww\4, wy+wh\2,
ww\2,wh\2, 1+512+1024+2048
```

```
' Print something novel in window 1
print " Welcome to HiSoft BASIC 2!"

' Set horizontal slider of the bottom window halfway
WINDOW CONTRL 1,0,500

' Set the horizontal slider size to be fairly small
WINDOW CONTRL 1,2,100
```

When you run the above program, it should look like this:




*Window handling in HiSoft BASIC*

Notice how this program works out what size its windows can be by obtaining these measurements from the free window that HiSoft BASIC gives you; this is good programming practice since it is resolution independent. The program will work under any screen resolution on any monitor. Please do not make assumptions about screen resolution in your programs - always get the size of the screen you are going to use by calling `WINDOW GET 2` (if you have kept HiSoft BASIC's free window) or `wind_get 0` in the *GEMAES* library (this gives you the 'backdrop' window sizes).



# Talking to the Outside World

It is probably true to say that taking in input and producing output, are the most important jobs that a computer program has to do.

There is very little value to line after line of complex calculation or data manipulation unless at the end of it all an answer is displayed on a page or the screen. We have already seen that without a `PRINT` command most programs are functionally useless.

Similarly it is certainly true that, with the exception of some graphics demos and the likes, most programs are useless unless they take some information from the user, a keypress or menu choice at the very least, or from the outside world via equipment such as a temperature sensor.

We have already looked at what are perhaps the two most fundamental commands that deal with input and output of information, `INPUT` and `PRINT`. There are however many different options that exist to complement or modify these basic keywords.

## Input and Output

There are several possible sources of information, not just data but also commands from the user, that are available for your program to act upon - these include the keyboard and mouse, files held on disk, and the computer's hardware ports. The options available for output include again the hardware ports, disk files, the speaker, a printer and, of course, the screen.

The operating system of the ST/TT computer itself can also often be looked upon as an external source of information for your programs as it acts as an intermediate between BASIC and the hardware.

The operating system is a special master program that is in charge of everything the computer does. It recognises what key you are pressing on your keyboard, it knows how to make the disk drives spin round when required and so on. It is this master program that also contains all of the GEM graphics routines.

The operating system keeps charge of the date and time clock that is available through commands such as DATE$, it is the operating system that provides information on the status of the disk drive (such as which files are present), it is the operating system that detects when many hardware devices signal an error (such as disk missing from the drive). All of these items of information can be used by your own programs to control which command lines and subroutines are to be executed.

Similarly when outputting data or instructions, the operating system may be the immediate receiver of the information you send. For instance when sending graphics commands your programs will be talking to GEM to get the desired effects rather than attempting to manipulate the screen display directly.

Each of these input/output 'targets' can be accessed by special related keywords or pre-defined library routines.

## The Keyboard, Joystick and Mouse

We have already looked at how the INPUT command works but although this is really a very simple keyword there are a few important options available.

You have the ability to suppress the question mark the command displays by the use of a comma instead of a semi-colon after the keyword.

The response given by the user must of course match the type of variable expected by the INPUT keyword or the user will get the message `Redo from start`. This is a wonderfully ambiguous message and means *'redo the current entry from the start'* (as if you could do anything else) rather than 'redo every input question you may have just been asked'. Strings entered as input need not be placed within quotes, and as long as they are not so enclosed the quote character - " - can be used as part of the string (but obviously not in the first character position).

Several items of data can be requested and entered in one go, although the number and type of the data entries must again match those required. In both the program line and during the entry process all items must be separated by commas.



LINE INPUT will read an entire line of data, until terminated by a carriage return, and assigns it to one string variable. It does this even when the information is separated into sections by commas which would have been taken to be data separators by the INPUT command.

If you have a line:

```
INPUT A$, B$, C$
```

and the user enters:

```
Bob Smith, 10 New Road, London
```

then A$ will equal `Bob Smith`

Whereas if you had a line:

```
LINE INPUT A$
```

which is given the same input A$ would be given the value of:

```
Bob Smith, 10 New Road, London
```

LINE INPUT is particularly useful when reading an ASCII file from disk, e.g. a file created by the HiSoft BASIC editor. Each line can be extracted from the file in turn and displayed, printed, or both regardless of whether it contains commas or any other form of punctuation.

Both INPUT and LINE INPUT allow the user to edit the data while it is being entered to make corrections by using the Backspace key to delete data already entered.

A closely related command is INKEY$ which reads the keyboard to see if any key, or combination of keys, has been pressed. If no key has been pressed an empty or null string is returned. The important differences from INPUT are that only one character value is returned at a time, and that the program does not have to wait for the user to hit Return before it can start to respond to the key.

If a key is pressed which is defined to produce a string of text, such as one of the function keys, INKEY$ will only take one character from the string every time the command is made.

INKEY$ is often used in conjunction with logical loops such as DO...LOOP to wait for a response from the user e.g.

```
SUB Wait
   PRINT "Press any key to continue"
   DO
   LOOP UNTIL INKEY$<>""
END SUB
```

The INPUT$(x) command is like a hybrid between INPUT and INKEY$ - it reads x number of keypresses from the keyboard and passes them directly to the program. This command is particularly useful as it allows a long string to be entered from the keyboard which need not be terminated by Return, and which can even contain, CHR$(13), in the middle somewhere.

Control of the mouse is via the keyword MOUSE This is very simple to use and there is a good example of its use in the *Command Reference* section.

The various joystick ports (the number varies depending on which ST/STE/TT computer you have) can be read by the following keywords.

Before trying to read the joystick(s) you must use:

```
STRIG ON
```

which turns off the mouse movement, and then:

```
STRIG OFF
```

when you have finished reading the joystick(s) and want the mouse back on.

```
STICK(n)
```

is a function that returns a reading of the position of a joystick position in x,y coordinates. The information the program requires from this function is signalled by the value of n which is passed. This can be 0 - 11 and the values returned respectively are the x co-ordinate of joystick 0, the y co-ordinate of joystick 0, the x co-ordinate or joystick 1 and the y co-ordinate of joystick 1 etc. The two ST joysticks are mapped to STICK(0) to STICK(3) whilst the extra STE joysticks are reached by STICK(4) to STICK(11).



When STICK(n) is requested, STICK(n+1) is automatically read as well; in fact you should always do a STICK(n) before doing a STICK(n+1) since STICK(n+1) does not re-sample the joystick position, it only recovers the value previously sampled by STICK(n).

```
STRIG(n)
```

returns a value as to whether any of the joystick buttons has been pressed; n can be from 0 to 7. Respectively these signal whether button 0 has been pressed since the last STRIG(0) command, whether button 0 is currently pressed, whether button 1 has been pressed since the last STRIG(2) command, whether button 1 is currently pressed etc. The buttons of the two ST joysticks are mapped to STRIG(0) to STRIG(3) whereas the STE joysticks' buttons can be addressed at STRIG(4) to STRIG(7).

## The Printer

The printer is not a standard peripheral for the ST/TT computer, but it is so common for people to buy one that it is given full support from the operating system and from within HiSoft BASIC.

The commands used for producing output on the printer are in many ways identical to those for laying out text on the screen, with the obvious exceptions that it is usually only possible to move the printhead *down* the page (some printers have a limited reverse paper feed option but it is rarely effective over more than a line or two) and that text can be overprinted, but not erased.

The LOCATE command is therefore of no use, and steps down the page are usually accomplished by sending a series of carriage returns, by sending special codes that force a printer 'line feed' (most machines allow you to set the line feed to be of variable height) or that cause the printer to perform a 'vertical tab' jump. If your printer supports either of the latter two options they will be explained more fully in the accompanying documentation

Some printers, especially those that are capable of different print pitches, can also use very many more characters across the page than is possible on the screen.

Within these constraints, and for the standard range of ASCII alphanumeric characters, you should have no problems in getting the correct output by using the keyword LPRINT everywhere that you would otherwise use PRINT.

However for any unusual characters, notably those in the ASCII range of approximately 128-255, which can be specified using the CHR$(x) command even if they cannot be typed directly at the keyboard, the effect produced on screen and that produced on the printer may be very different. Daisy wheel printers may be incapable of producing any output at all when sent such characters.

Conversely many printers are capable of producing effects, styles and font types that are impossible for the screen to use when printing text (at least without first going through GEM display controls first) such as underlined, italic, bold or condensed text. These are triggered by the use of a certain control sequence of ASCII characters - your printer manual will provide precise details for your type of machine. Similar codes are also used to cause the printer to feed the paper through by one line, or by a whole page and so on.

Graphics printing can be done on many printer types but the process by which an image is produced on paper is fundamentally different to that which is used on the screen. Again your printer manual should give details if your machine is capable of producing such output. Unfortunately not all printers agree on which codes should be used to control this graphics output and if you are writing software you intend to sell you should try to support a range of types, in particular machines that conform to the Epson or IBM graphics standard.

The LPOS keyword returns the value for what the computer feels should be the horizontal position of the print head on the paper, based on how many characters have been sent to the printer since the last carriage return. The situation can actually arise where the computer gets this wrong, e.g. if the printer head was left in an unusual position by a previous program. The keyword is also unable to keep track of what effect, if any, the tab character is producing on the printer (the size of tab jumps can be defined on most machines) but on the whole the information returned should be reliable.

LPOS is typically used for testing whether there is enough room on the line remaining to send the next word you wish to print. If there is not, a carriage return/line feed can be sent first ensuring that no words are broken in half.

Because of the relative lack of control you have over this feature there is no keyword for returning the vertical position of the print head.



As with text printing on screen the WIDTH LPRINT command can be used to set the maximum width of the printed display. After the specified number of characters have been sent a new line is automatically started.

## The Hardware Ports

The expansion ports of your ST/TT perform a variety of jobs which are principally to do with the transfer of information between the computer and a range of peripherals, other equipment, or other computers.

For the most part the only peripherals that are of particular interest to 99% of computer users are the keyboard/mouse/joystick and the printer. To reflect the frequency with which these essential pieces of equipment are used there are a variety of pre-defined keywords that allow the user to access them and which we have already covered.

Communication with less standard equipment can be rather more complex. Because no friendly pre-defined commands exist to anticipate connection with, say, a temperature sensor or a bar code reader we have to resort to a series of keywords that send raw information to the various ports. You will have to study the hardware documentation of the equipment to ensure that you can work out what types of electronic messages it will expect from the computer, and what type of signals it will send back. You will also need to know how to signal when one piece of data has finished and another has begun and so on.

If you have unusual equipment to drive this is the perfect opportunity to try your hand at writing some procedures.

The building blocks we have available are INP() and OUT(). Within the brackets you must put the number of the expansion port that you wish to read information from, or send information to, respectively.

The following expansion port numbers are available on the ST:

0 - the parallel or Centronics port

1 - the serial or RS232 port

2 - the standard input/output device

3 - the MIDI interface

4 - the keyboard controller, directly

5 - direct screen output

The 'standard input/output device' is computer talk for the keyboard for input, and the screen for output. Unless you have certain specific requirements in mind, such as tinkering with the screen display, it is unlikely that you will use this option.

The parallel and serial ports are most often used for communicating to printers and plotters, although the latter is also important for use with modems for accessing 'electronic mail' systems such as CIX. Again, unless your requirements are unusual, it is unlikely that you will use these options for printing as the simple LPRINT command will suffice. Attempting to drive the ports directly will open up all sorts of cans of worms which would otherwise be conveniently hidden away by the operating system.

The serial port in particular needs to have certain parameters set which tell it amongst other things the speed at which data is to be sent down to other peripherals, and the speed at which returning information is to be expected to arrive. These figures are expressed as the *baud rate*. The baud rate is a measure of the number of bits of information sent each second (remember that a 'bit' is a precise measure of information in computing terms).

Other terms you will come across include 'stop bits', 'handshaking', 'duplex' and 'parity'. It is not really necessary to try to detail the meaning of these terms here - the manual for the peripheral you wish to use will tell you what settings each of these features should have. Specialist books on computer communications are available from most suppliers.

HiSoft BASIC requires you to make calls to the operating system through the BIOS Library to make changes to any of these settings. These Library routines are documented in the *Reference Manual : Chapter 2*.

The MIDI interface is designed for communicating with electronic musical instruments. Again support for this device is a fairly specialist technique and you are advised to look for texts dedicated to this subject.



It is difficult to provide demonstration programs for these keywords, since they will only be applicable for a given piece of hardware.

## Sound

HiSoft BASIC provides many ways of handling sound on your ST/TT computer. First of all, here is some (very) simple keywords.

We have met BEEP already and there is really nothing more to say about it as it is a very simple command. It is most useful as a means of signalling an error or attracting attention to a message in your program. It is worth noting that a similar effect can be achieved by the command:

```
PRINT CHR$(7)
```

This can be useful when you have written a routine that reads a series of ASCII numbers, say from a file held on disk, and prints them onto the screen. ASCII number 7 can be held within the file to signal the end of a particular message or something like that.

The command SOUND is only slightly more sophisticated. It allows the pitch and duration of the note to be varied, and is detailed in the *Command Reference* chapter.

The WAVE command is a useful complement to the keyword SOUND as it allows the type of noise that is to be produced to be controlled by defining the waveform and envelope of the noise. This is again explained in more detail in the *Command Reference* section.

There is also a range of STE sound commands to be found in the supplied libraries - see *Chapter 2* of the *Technical Reference* manual for more detail.

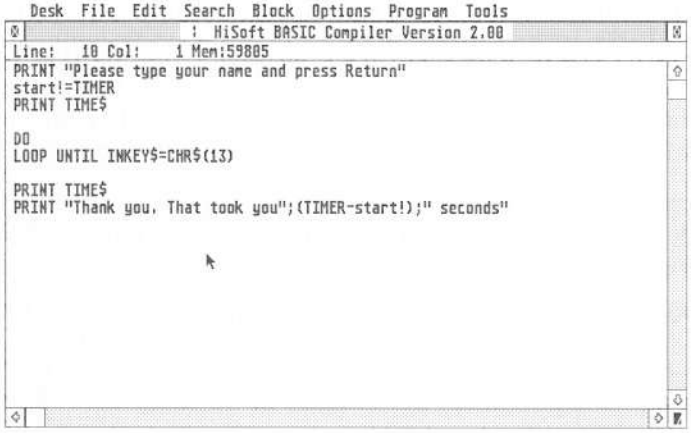
## The Timer

The timer is not a true physical device on the ST/TT computer, but the effect of one is simulated by the operating system as a constantly active background task. As long as the machine is not turned off or reset the current date and time can be read or reset through HiSoft BASIC.

DATE$ and TIME$ can be used as functions to return the current value of the 'clock' settings, or can be assigned new values as if they were predefined variables. The values are returned as specially formatted strings.

TIMER is a function that returns the number of 200ths of a second that have elapsed since the last call to it. The returned value is a single-precision numeric value so two calls to the TIMER function can be used to time a given part of the program with much more accuracy than using the TIME$ command (TIME$ returns information as a specially formatted string). TIMER's accuracy is to the 200th of a second, whereas TIME$'s accuracy is 2 seconds.

Try running this program to see the difference between TIMER and TIME$:




*TIMER and TIME$*

## Managing Files

HiSoft BASIC provides a complete suite of commands to allow the programmer to access and control the filing system. All large utility programs you write should employ these commands to allow the user to best organise the layout of files on the disk from within your program, particularly if they may need to create room for data by erasing or copying some of the existing titles.



FILES will produce a list of all files present in the current default drive and sub-directory. The command can be followed by a string that will allow the displayed filenames to be limited, relative to the entire list, on the basis of what are known as filename 'wildcards'.

The two wildcard characters available are * and ?. The former is a shorthand that stands for 'any string of letters'. The latter stands for 'any letter'.

Try:

```
FILES "*.*"
```

you should get a list of all files on disk.

Now try:

```
FILES "*.BAS"
```

you will get a list of only those files that have the extension .BAS.

```
FILES "D???.BAS"
```

will list all BASIC files with four letter names that begin with D and so on.

The filename used by the FILES command can also include drive letter and directory name extensions (see below).

```
KILL filespec
```

will delete the named file specification from the disk. Again wildcards can be used in the filename which, unless you are very careful, is the road to disaster.

```
INPUT "Which file(s) do you want to erase"; A$
KILL A$
IF A$="*.*" THEN PRINT "Now go and shoot yourself!"
```

It is recommended that you write your routines such that they first get a list of files using the same wildcard name. This should then be followed by a request for confirmation before deleting anything in order to check that you will not lose anything vital.

NAME filename1 AS filename2

allows a specified file to be renamed. `filename1` receives the name that is specified as `filename2`; note that you can not swap names of files or have two files with the same name.

CHDIR

changes the default directory for the current drive. This is helpful when using the FILES and NAME commands, and for using any newly created or modified files (see OPEN, BSAVE etc. later on).

Your ST manuals will explain how the directory system works, but briefly: each disk drive you have (including perhaps a simulated disk drive created in the memory by ramdisk software) will be referred to by its drive letter - starting with A:, then B:, C:, D: etc.

On each disk you can store files in one large unstructured group, or choose to organise them into named directories. A directory is a type of filing system folder in which you can store those files that are pertinent to a certain subject. Each directory can have several sub-directories which again can be used to divide the information up into useful groups. Files with identical filenames can be stored in two different directories without causing any problems.

For example you can have on drive B: a directory called ACCOUNTS which can in turn have two sub-directories - OFFICE and HOME. A file called LETTER in the latter sub-directory can be accessed by the pathname:

```
"B:\ACCOUNTS\HOME\LETTER"
```

The pathname of the file tells the computer the route it will have to take through the various directories to access the data.

Using the CHDIR command you can make the sub-directory `B:\ACCOUNTS\HOME` the current directory. Files held there can then be accessed by their short name only and commands such as FILES will operate on that specified sub-directory if just a filename is used.

MKDIR pathname

creates a new sub-directory from within the current default drive and directory or via a specified pathname.



RMDIR pathname

removes an existing named directory as long as it contains no files at the time.

Other disk drive operations, such as formatting a disk, can be controlled by calls to the XBIOS library that is supplied with HiSoft BASIC.

## Reading and Writing Disk Files

All computers need a system for permanently storing the information that the user has entered, whether this is a program, a list of names and addresses, a screen picture or whatever.

Manipulating data files on disk is one of the most important jobs your computer can do. Data storage is increasingly what computers are about, as the development of appropriate hardware such as hard disks drop in price so computer languages are developing to make it easier for the user to access and use data held in a permanent record. There are however different types of disk file that can be used, each appropriate to different circumstances.

The simplest type of disk data file you can create is a completely unstructured record of the contents of a specified part of the computer's memory. The most frequently used example of this technique is when a graphical image on the screen, made up from a pattern of dots held in memory, is saved for later recall. The way screen pictures are stored and the implications of this technique are detailed in the section on *The Screen*.

The keywords we use for writing and reading simple memory-image files are BLOAD and BSAVE.

To understand how these commands work we must learn something about how the computer's memory is structured.

Every item, or byte, of data (which is large enough to store one single letter of a text string for example) is stored at a specified memory address. The amount of memory available varies - on a 520ST the number is 512*1024 bytes (a kilobyte is 1024 bytes rather than a thousand, because of the way that computers work internally with binary arithmetic).

Not all of memory will be free to use for data - many locations store important working space for the operating system and the programs that are currently loaded.

The syntax for BSAVE and BLOAD is:

```
BSAVE "filename", start_address, length
BLOAD "filename", start_address
```

The filename can contain a drive letter and sub-directory path.

The restrictions on the use of filenames is rather complex (see your Atari Manual for an exhaustive list) and it is important to ensure that your program contains a suitable selection of routines to trap nonsense entries by the user.

The most common use for BSAVE and BLOAD is to save and re-load data from that portion of memory that contains the screen display. There is a slight complication in that the actual addresses used to hold the screen data are not fixed on the ST, but can be re-located by a special call to the operating system. This is covered in more detail later in the manual but here's a quick example:

```
PRINT SPC(20);"HiSoft BASIC 2"
REM Save the entire screen
BSAVE "SCREEN.DMP", PEEK(&H44E&),32000
CLS
PRINT "Now reloading ..."
BLOAD "SCREEN.DMP", PEEK(&H44E&)
DO
LOOP UNTIL INKEY$<>""
```

As an alternative to memory-image files it is also possible to define much more highly structured files that hold text and numeric data in special formats such that the computer can at the very least distinguish the length of each entry and its position within the file. The data within such files consists of individual items, normally known as *records*. It is possible to add or extract specific items of information from these files or to add data items of data to them.



There are actually two types of structured file you can use with HiSoft BASIC. The first is known as a *sequential file* and it holds data that the computer can only access by stepping through item by item in sequence. For example, you can read the twentieth record in the file only by first reading records 1 to 19. The advantage of such a sequential system is that each individual item of data can be of any length or form that you desire, provided that the end of that particular item is signalled somehow, whether by a comma, by quotemarks ("), by carriage returns, or a combination of these.

The second type is a *random access* file. This uses a system whereby each record of data is of a preset and rigid length and form (and we will see later that any data that is not of the right type has to be converted to match) such that the computer can automatically calculate the position of item number 20 and jump straight there - hence the name random access.

Because random access files contain items of a preset length, old data can safely be overwritten by new without any danger of obliterating the next entry. Sequential files can only have new data added to the end of existing files. To remove or insert existing items, an old file must be copied item by item into a new file, with the appropriate modifications made during the process.

Random access files are therefore often very much faster in practice (although this does depend on the use to which you are putting them). They have the disadvantage that they can be more wasteful of disk space - all records must be allocated the same room regardless of their actual length; this in turn is determined by the longest entry you have to accommodate.

The type and structure of a data file is defined when it is created, and this is done by the OPEN command. It is no surprise then that this can be a rather complicated keyword with several parameters:

```
OPEN file_spec [FOR mode] AS [#]channel_num
[LEN=record_size]
```

HiSoft BASIC can have 255 files open at once, so it is fundamental that each of these files is designated by a number as it is opened. Any future disk reading or writing commands can then be told the file they are to work on by referring to that number. Assigning the number is done by using the AS # part of the OPEN command.

A simple

```
OPEN "filename" AS #n
```

command will create a new sequential access file, or open an existing one of the given name for reading and modification.

However it is possible to define the particular type and nature of the file that has been opened using the following commands:

```
OPEN "name" FOR OUTPUT AS #2
```
a sequential file for writing.

```
OPEN "name" FOR INPUT AS #9
```
a sequential file that is to be read.

```
OPEN "name" FOR APPEND AS #1
```
a sequential file that already exists and is to have data added to the end.

```
OPEN "name" FOR RANDOM AS #5
```
a random access file

The final option is `LEN=number`.

This sets the length of each entry in a random access file. The default size, if the command is omitted, is 128 bytes long. If this command is used for sequential files, it controls the size of the memory buffer that has to be filled before any data actually gets written to disk. Disk operations such as writing data are, in computer terms, relatively slow. The ST/TT very sensibly waits until it has a reasonable amount of data to write before it goes to the trouble of moving the disk drive heads around etc.

## Sequential Files

Putting data into a sequential file, and retrieving the data from such a file, in many ways parallels the system used for reading information from the keyboard, and writing to the screen or printer. Indeed so similar are the operations that the keywords used are almost identical.



`INPUT#` and `LINE INPUT#` allow data to be read from a sequential file. As with input from the keyboard, `INPUT #` reads data that is to be assigned to a specified list of variables. `LINE INPUT#` reads an entire line of a sequential file until it reaches a carriage return character, ignoring any commas of other data delimiters and assigning the result to one string.

`INPUT$(x)` can read `x` bytes of data from either a sequential or a random access file regardless of any data separators or field boundaries it crosses in doing so.

As we have seen, putting data into an existing sequential file can only be done by adding it to the end of those records already saved. The keywords to use to do this are `WRITE#`, `PRINT#` and `PRINT# USING`.

As with output to the screen, `WRITE#` automatically separates different data items with commas, and encloses strings within quotes. The full options available with the `PRINT#` and `PRINT# USING` commands, and the others, are detailed in the *Command Reference* chapter. When used to write data to a file they create an exact image of the information that would be seen on screen so certain mistakes are possible if you are not careful.

For example:

```
PRINT#1, 1,2,3,4,5
```

will assume that the commas are to be replaced by spaces as on the screen. If the items were intended to be stored as separate data items the commas have to be `PRINT`ed explicitly as in:

```
PRINT #1, 1;",";2;",";3;",";4;",";5;
```

`PRINT #` is probably best used for producing formatted lines of text that are to be retrieved by the `LINE INPUT#` command, or for storing formatted numbers as produced by `PRINT# USING`.

Note the distinction between a `PRINT`ed, and therefore formatted, line of text, as would be produced by a word processor for example, and a structured text file such as may be produced by, say, a database using the `WRITE` command. In the former case the output will look more meaningful to an observer, but the computer itself will be unable to determine what each individual item of data represents, and where one data entry ends and another begins. In the latter case every item of data is clearly separated from the next by commas and quotes.

The WIDTH# keyword can be used to limit the maximum length of any one line of data that is PRINTed to the file.

Try the following example:

```
A$ = "John Smith"
B$ = "21 New Road"
C$ = "London"
OPEN "WRITE.DAT" FOR OUTPUT AS 1
WRITE #1, A$, B$, C$
OPEN "PRINT.DAT" FOR OUTPUT AS 2
PRINT #2, A$, B$, C$
CLOSE #1, #2

REM Now read using LINE INPUT
OPEN "WRITE.DAT" FOR INPUT AS 1
OPEN "PRINT.DAT" FOR INPUT AS 2
LINE INPUT #1, A$
LINE INPUT #2, B$
PRINT "WRITE data looks like:"
PRINT A$
PRINT : PRINT "PRINT data looks like:"
PRINT B$
CLOSE
```

The file called WRITE.DAT will have the data stored internally like this:

```
"John Smith","21 New Road","London"
```

The file called PRINT.DAT will have data stored internally like this:

```
John Smith    21 New Road    London
```

Any data stored using the PRINT # keyword can be formatted within the file in the same way as it can on the screen:

```
PRINT #2, A$; B$; C$
```

would produce a file containing

```
John Smith 21 New Road London
```



whereas:

```
PRINT #2,A$
PRINT #2,B$
PRINT #2,C$
```

will produce a file like this:

```
John Smith
21 New Road
London
```

Of course each of these different formats can be accessed in different ways with the INPUT# and LINE INPUT# commands. For example, using the first example file produced with the WRITE# command:

```
OPEN "WRITE.DAT" FOR INPUT AS 1
INPUT #1, A$
PRINT A$
```

would produce the result:

```
John Smith
```

whereas:

```
OPEN "WRITE.DAT" FOR INPUT AS 1
LINE INPUT #1, A$
PRINT A$
```

will produce the result:

```
"John Smith","21 New Road","London"
```

The LINE INPUT# command has ignored the data separators and has read a whole line from the file into the variable A$. Conversely in the case of the first PRINT example:

```
OPEN "PRINT.DAT" FOR INPUT AS 1
INPUT #1, A$
PRINT A$
```

would produce the result

```
John Smith     21 New Road     London
```

The INPUT # keyword has found no recognisable data separator to denote where one entry finishes and the next begins and:

```
OPEN "PRINT.DAT" FOR INPUT AS 1
LINE INPUT #1, A$
PRINT A$
```

would produce the same result.

Obviously, different techniques are required depending on how you choose to store data from within your programs.

## Random Access Files

When using random access files things are rather different. The rather inflexible format with which these files are stored requires a much more complex system of data manipulation before the information can be stored on, or retrieved from, disk.

To recap, random access files are divided into records, but these are of a fixed length, defined at the time the file is opened, and can only contain data in a fixed format (actually stored as binary data) whether it started off as text or numbers of any precision. Many different keywords are therefore required to convert the data into the specified length and the required form.

Random access files have one or two other important traits. Firstly the data held within a given record can in, turn, be subdivided into a series of fields each of which hold a subset of information that is pertinent to the whole record. For example, in a club mailing list file, each member would merit a record of their own, but within that record there will probably be a field for NAME, one for ADDRESS, one for TELEPHONE NUMBER etc.

The keyword FIELD is used to allocate space within the total record to each sub-division:

```
FIELD 1, 20 AS name$, 50 AS address$, 12 AS tele$
```



Each string variable that is defined in the FIELD statement is called, of all things, a *fielded variable*. Fielded variables are rather special and should only really be used in conjunction with the RSET and LSET commands that we have already met, in order to produce a precisely-formatted string. Data can not be allocated to these variables using other commands such as INPUT or they will lose their 'fielded' status.

A FIELD statement must be issued before reading or writing a random access file. Even if the records are not subdivided, at least one field string must still be defined to hold the data that is to be written to the file.

Because all data held within a random access record has to be of the defined length then if it is too long it will be truncated and if it is too short it has to be padded out to fit. To do this all data is converted to a string form of the required length, before being saved. To recover it again it has to be converted back from a string to the data form it originally started with.

The process of writing data to a random access file is therefore as follows. Firstly all numeric data has to be converted to a string. The following keywords do this job:

| | |
|---|---|
| MKI$(x) | converts an integer x to a string. |
| MKL$(x) | converts a long integer x to a string. |
| MKS$(x) | converts a single precision number x to a string. |
| MKD$(x) | converts a double precision number x to a string. |

The next step is that the resulting string has to be placed in the random access buffer, an area of memory automatically reserved as workspace when the file is opened, ready for writing to the disk. As this is done the data is padded to the length required to fit either the whole record or a field to which it is to be assigned. This is done through the use of the keywords LSET or RSET. LSET fits the given string to the left hand end of a padded string of the required length. RSET fits it to the right hand end of a padded string of the required length.

For example the command:

```
RSET A$=MKL$(x)
```

will fit the string equivalent of the long integer variable x, into the right hand edge of the fielded variable (A$) ready for storage.

If the name of the variable, A$, has been designated as a field occupying a subsection of the total record length (see above) the computer will automatically store the data in the correct place within the total buffer. In this case RSET or LSET will place the data into the right or left hand end of the room allocated to that variable.

Remember that MKtype$ is only necessary when storing numeric data. Commands such as:

```
RSET A$=B$
```

or:

```
RSET A$="Fred Smith"
```

can be used to place text data directly into a fielded variable.

When all of the fielded variables have been assigned the appropriate data, or left blank if required, the buffer memory can be copied onto the actual disk.

The command required is PUT#n, x, which writes the current value of the buffer data to the correct place within file number n, as record number x. If x is omitted the next record in the file is written. The buffer is 'emptied' as a result of this command so that the next record to be written will be starting with a clean sheet, as it were.

That was many new concepts to take in, but the process is really very simple if you take it step by step. Here is a simple example:

```
A$="John Smith"
B$="21 New Road London"
X%=21
OPEN "PERSONAL.DAT" FOR RANDOM AS #1 LEN=65
FIELD 1, 20 AS name$, 40 AS address$, 5 AS age$
LSET name$=A$
LSET address$=B$
LSET age$=MKI$(X%)
PUT 1,1
```



Retrieving data from a random access file is a slightly simpler process. The opposite of the PUT command is GET. Before issuing this command, the field variables should again have been declared in the program. Once one record has been read these fielded variables can then be used immediately, for example to print individual sections of the data, or the retrieved data can be assigned to other variables for manipulation freeing the fielded variables for reading another record.

If the data that has been retrieved was originally assigned to a numeric variable it can be converted back to its original form using one of the following commands.

| | |
|---|---|
| CVI(string$) | converts the data back to an integer. |
| CVL(string$) | converts data back to a long integer. |
| CVS(string$) | converts data back to a single precision number. |
| CVD(string$) | converts data back to a double precision number. |

Note that these commands work in a very different way to the VAL keyword, and they cannot be used in its place. Note also that random access files require that you can anticipate the likely precision of numbers before they are converted. Sequential files are more tolerant in this respect - a numeric variable of an unspecified type can be used to hold a given value and that variable will to an extent adjust its internal format to match the data it is assigned.

```
OPEN "PERSONAL.DAT" FOR RANDOM AS #1 LEN=65
FIELD 1, 20 AS name$, 40 AS address$, 5 AS age$
GET 1,1
PRINT name$; CVI(age$)
```

will produce the output

```
John Smith      21
```

Note that the variable name$ is padded with spaces to the length defined in the field statement; you can use RTRIM$ to cut this data down to size.

Finally we have the keywords LOF, LOC and EOF which are three functions that allow the program to keep track of the size of a file and the current position of the pointer within it.

LOF stands for Length Of File (in bytes).

LOC stands for the LOCation of the data pointer. When using random access files it reports which number entry was read by the last GET command or was last written to by PUT. For sequential files LOC is often less useful. It returns the current byte position of the file pointer divided by 128. If your records are of a set length the actual record pointed to can be calculated from this value even if your data entries are longer or shorter than 128 bytes.

EOF, standing for End Of File, is an invaluable logical function that reports TRUE (-1) when the data pointer has reached the last data entry. It is usually used in logical loops to trigger the end of a certain operation as in this example which reads an ASCII file and copies it to the printer:

```
SUB Print_File
   FILES
   INPUT "Which file do you want to print?"; A$
   OPEN A$ FOR INPUT AS #1
   WHILE NOT EOF(1)
      LINE INPUT #1,aline$
      LPRINT aline$
   WEND
   CLOSE #1
END SUB
```

Shutting down a file is a very simple affair - just use the command CLOSE for either sequential or random access data files. A list of numbers can be given to close several files at once, and the keyword CLOSE without any specified numbers shuts down *all* files.

The keyword RESET has the same effect. All FIELD statements are forgotten at the same time. The various program termination commands STOP, END and SYSTEM also automatically shut down all files.

All OPENed files must be closed if the disk is to be removed while the program is running or data may be ruined. At the very least any data still held in memory buffers will be lost. Because of the threat of power cuts you can consider including a routine in all your programs that, perhaps at a timed interval, makes a backup copy of the current files.



The final command that may be of value is the keyword VARPTR#. This will return the address of the buffer used to store information that has just been read from, or just prepared for sending to, the file number. This keyword is principally designed to pass information to routines written in other languages such as assembler to allow them to access the data held by your HiSoft BASIC program.

## Error Handling

Error Handling is a means whereby the programmer can anticipate, and make allowance for, undesirable circumstances whilst the program is running. These 'errors' are of three main types. They may result from a problem in the hardware such as no paper in the printer, no room left on the disk or no disk in the disk drive. They may follow an incorrect data entry or response by the user. Finally they may be triggered by some error in the programming that did not become apparent under test.

HiSoft BASIC, in conjunction with the ST operating system can pick up, and determine the nature of, a wide selection of errors and it is good practice that your program should try to prepare for as many of these as possible. Error trapping routines can be tedious to plan and write, and can seem to take a disproportionate amount of time compared to the real nuts and bolts of your program, but if done well they are the hallmark of a professional quality program.

We have already seen how it is possible to enter simple error traps by specifying certain logical conditions that accept or reject data entered by the user. For example:

```
DO
   LOCATE 1,2
   INPUT "Size of curve (1-4) ",s%
   y%=length/s%
LOOP WHILE s%>=1 AND s%<=4
```

This continues to ask the question until the user puts on his spectacles and provides an acceptable answer. Note also that the the variable s% is an integer type so that an entry such as 1.4 would be converted to a usable number.

However, it is part of the nature of error catching routines that there are always more possible ways that the user can catch you out than you will ever anticipate in the first attempt; let's say he enters the value 0 - you've forgotten to trap this and you will get a runtime error on the fourth line because of division by zero.

It is also inevitable that 'errors' can occur that you will be unable to prevent happening, such as an unformatted disc being placed in the disc drive and so on. We therefore need a system of dealing with such errors no matter when and how they occur.

Errors that are produced through a problem in the hardware or through a mistake in the program, such as trying to divide an expression by a variable which has somehow been assigned a value of zero, will generate an error number specific to that particular situation.

To allow the maximum use of this facility there are a range of functions that will report the nature of the error, together with the line number which was being executed when the problem occurred. ERR is the function that returns the error code and ERL returns the line number where it happened. From these your routines can deduce what the problem was and decide how to fix it (e.g. print a message such as `There is no disk in the drive` and then restart the data input routine).

The nature of errors means that they can only be anticipated to occur at un-anticipated times. Similarly the response to the error has to be ready to come into operation at any time. Hence error handling really has to be organised as a background event trap.

The command sequence we use is therefore is ON ERROR GOTO which causes a jump to a specific line number or line label as soon as any error occurs. The line that is jumped to will usually be the beginning of a routine that decides exactly which error has occurred, deals with it in someway, and decides where (if at all) that the program is to resume execution.

The keyword RESUME specifies which line number or line label the program is to return to following the specific error correction routine has been completed.



Here is an example:

```
a$="TEST.DAT"

Kill_File:

ON ERROR GOTO Kill_Problem
KILL a$

ON ERROR GOTO 0     ' Normal error reports

STOP

Kill_Problem:

ErrNum=ERR
REM Can't use ERR directly in CASE

SELECT CASE ErrNum
'Error 53 is File not Found
   CASE 53
      FILES
      PRINT a$;" not found"
      INPUT "Please give a valid file";a$
CASE ELSE  PRINT "Fatal error ";ERR : STOP
END SELECT
RESUME Kill_File
```

As an extension of the error facility the programmer can also add to the list of errors that will be reported by specifying new conditions that are to be regarded as problems of some kind. The keyword ERROR can be used in this context to define the error that is required. It can be used to extend the range of conditions under which an existing error number is triggered:

For example: runtime error 5 is `Illegal Function Call` - you might want to use that in one of your own user-defined functions as shown on the next page:

```
REM Factorials by iteration

FUNCTION Factorial#(x)
STATIC Temp#,i

REM Can't do 0! or 1!
   IF x<2 THEN ERROR 5

   Temp#=1
   FOR i=2 to x
      Temp#=Temp#*i
   NEXT i
   Factorial#=Temp#
END FUNCTION
```

The ERROR keyword can also be used to define the conditions that would trigger an error number that has not previously been used. This is invaluable when using unusual peripherals for which there are no pre-defined errors, and can even be used to catch situations such as when a data input has exceeded a permissable maximum, or is not of the appropriate format etc.

```
Get_Input:

ON ERROR GOTO Validate
INPUT "Choice (1-4)";Choice
IF Choice<1 OR Choice >4 THEN ERROR 200
INPUT "Name, please"; Name$
IF LEN(Name$) > 19 THEN ERROR 201

REM Normal error reporting
ON ERROR GOTO 0

Validate:
ErrNum=ERR
SELECT CASE ErrNum
   CASE 200
      PRINT"Please select a number between 1 and 4"
   CASE 201
      PRINT "Names must be less than 20 characters"
   CASE ELSE PRINT "Unknown validation error" : STOP
END SELECT
RESUME Get_Input
```



This routine most usefully works in conjunction with an ON ERROR statement as the newly defined number can then, if triggered, cause a jump to a sub-program which will print an explanatory comment. If error handling is not enabled the effect of the newly defined error, when triggered, is to print the message Runtime error nn where nn is the error number.

# The HiSoft GEM Toolbox

The *HiSoft GEM Toolbox* consists of a number of modules, written in HiSoft BASIC, that allow easy high level access to the GEM graphics system that is supplied with all current Atari 680x0 computers. Each module has a particular function so that you can build up your programs in a modular way.

The source code of each of the toolbox modules is supplied, together with a number of pre-tokenised combinations. Normally a toolbox routine is 'included' by the program that uses its sub-programs and functions but a faster compilation will result if you use a pre-tokenised version.

Full details of the *HiSoft GEM Toolbox* can be found in *Chapter 3* of the *Technical Reference* manual.

In this section, we will take you through the rudiments of designing and writing programs using the toolbox.

## Your first program

As a way of introducing you to the power of the *HiSoft GEM Toolbox* (shortened to HGT from now on) here is a practical example of using the toolbox. Remember that you will need 1Mb of memory or more in order to work with HGT.

We will present a slightly arbitrary program that is, none the less, a good example of the principles of GEM programming with HGT. The program opens a window, fills the window with text, adds a menu bar and responds to two menu items: Quit the program and About the program. This can all be achieved in a remarkably short space of time (and code) using HGT.

The main steps in producing this program can be summarised as follows:

- Design the GEM objects, choosing logical names for the objects that will be referred to from the BASIC program.
- Code the outline of the program i.e. the sub-program and function names and the main code so that the structure is easy to understand and to maintain.
- Write the code for any sub-programs and functions.
- Test and debug the program until it works to your satisfaction.

## Designing your GEM objects

Firstly, we must design our menu and the dialog boxes that will appear when the menu items are selected. This is a job for WERCS, the resource editor, since it would be very tedious to create these items using BASIC alone.

It will be useful to keep all your preliminary HGT work in a folder so create a new one on your work disk or hard disk called TUTORIAL. We will keep all our HGT files in this folder.

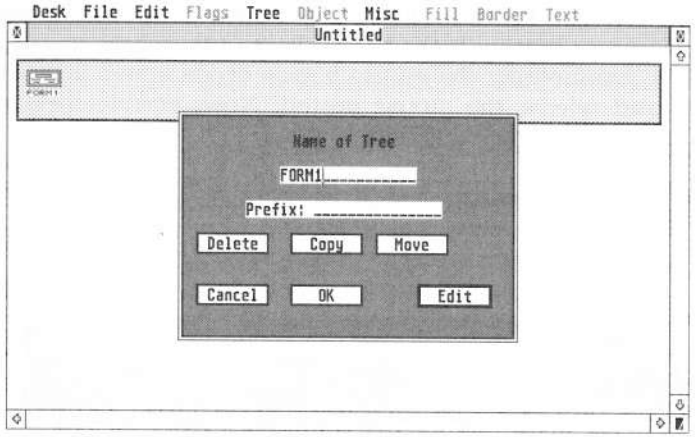
Run HiSoft BASIC and set the current directory to your TUTORIAL folder - do this by selecting Change Directory on the File menu, choosing the correct folder and clicking OK. The editor will now use this path for your source code and for any tokenised files.

To run WERCS, move up to the Tools menu and select WERCS; the resource editor should appear.

When it appears, move the mouse pointer to the Tree menu and select Form; we are going to design a dialog box first.

> Normally, an About box would be an alert, not a dialog box, since it does not need any user interaction. We are using a dialog box here deliberately to give you practice.

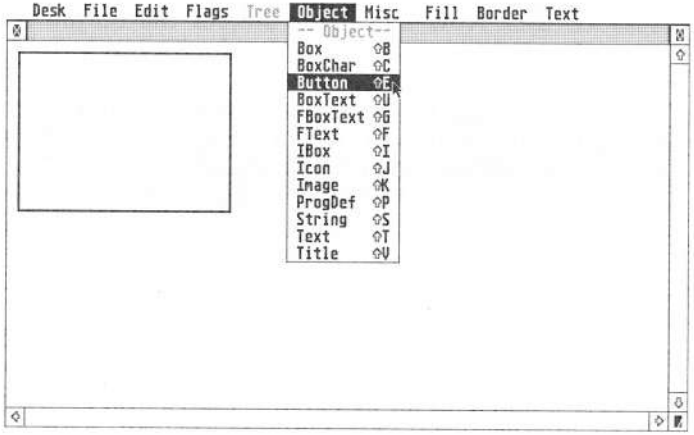





*Naming the About dialog box*

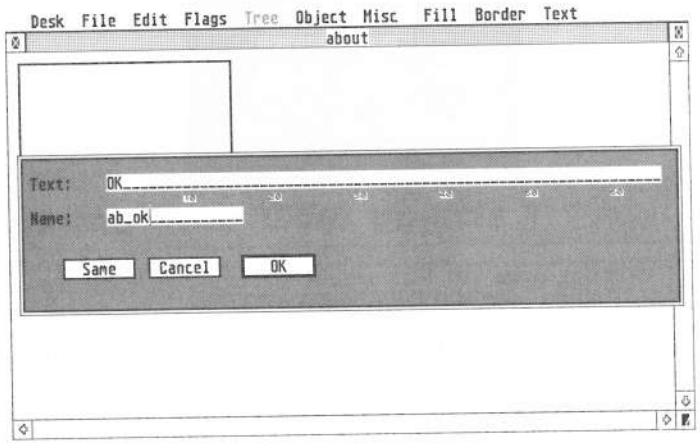
You may find it useful to refer to the WERCS section of the manual during this part of the tutorial.

Press Escape to clear out the name text and enter a suitable name for this dialog box, say `about`; now click Edit or hit Return. Select Button from the Object menu since we are going to add a button to the dialog box.




*Adding a button*

Move the OK icon towards the bottom centre of the dialog box and press the mouse button. If the button isn't quite where you want it, click on it and, holding the mouse button down, move it to a better position and release the mouse button. Now, double-click on the button to edit its text.

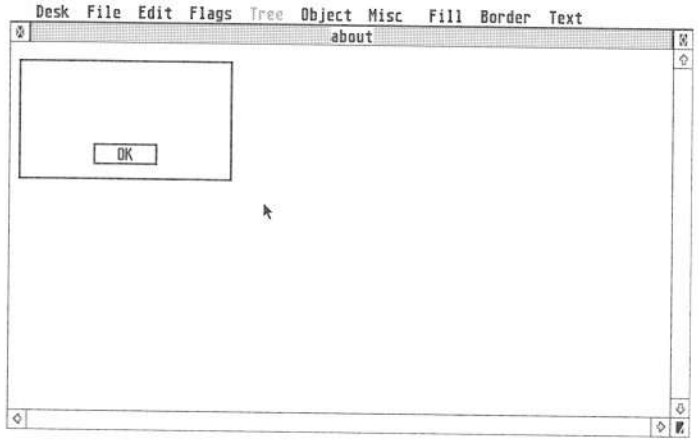



*The text of the button*

Press Escape, enter OK and then press the Tab key to move to the Name field; enter a sensible name for this object, ab_ok. Click on OK or press Return. Now adjust the size of the button and the outer box until you have something that looks good; do this by clicking on the bottom right of the button or box, holding the mouse button down and moving the mouse until the size looks right and then releasing the mouse button. If you miss and select the wrong object, just click outside the outer box to de-select it.

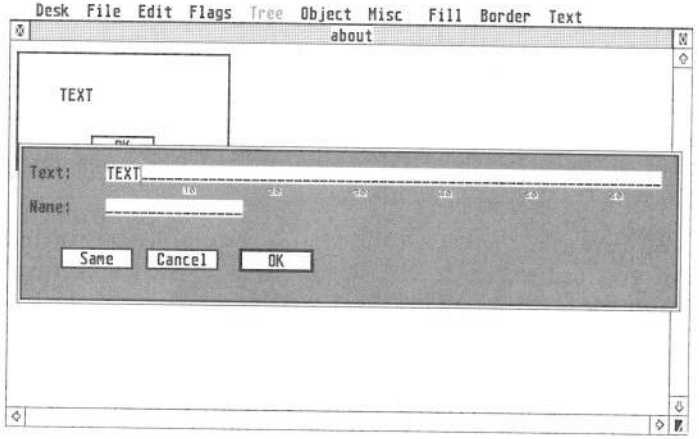


You should end up with something that looks like this:




*The button placed in the dialog box*

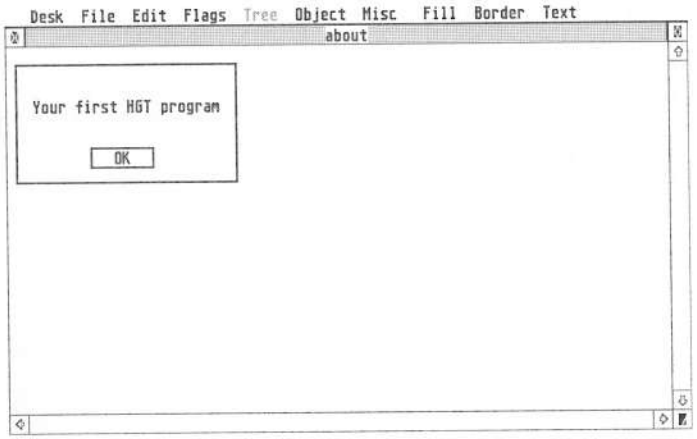
Next select Text from the Object menu and place the TEXT icon towards the top left of the dialog box to add some words here. Double-click on it once you have placed it and the following box will appear:




*The title text of the dialog box*

Press Escape, type Your first HGT program, press Tab and name the text ab_text; click OK or press Return. Place the text (by picking it up with the mouse) in the top centre of the box.

The completed dialog box should look like:




*The completed About box*

Click the *Close* box at the top left of the window.

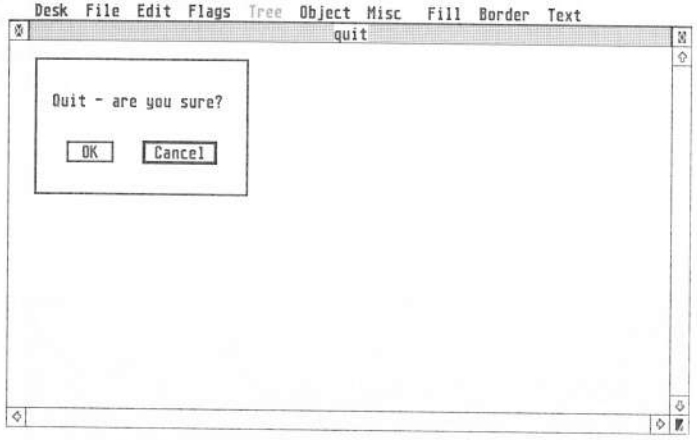
You have now created the About dialog box so let's save our work before continuing. Select Save from the File menu - a file selector will appear; hit Escape to clear out any filename already present, ensure that the path (where you are going to save this resource file) is your TUTORIAL folder on your work disk and then type `HGT1.RSC` and click OK.

This should save your resource file under the name HGT1.RSC, it will also create two other files called HGT1.HRD and HGT1.BH, more of which later.

See if you can now create a Quit dialog box yourself, in the same way that you put together the About box; start by selecting a new Form from the Tree menu and call it `quit`.



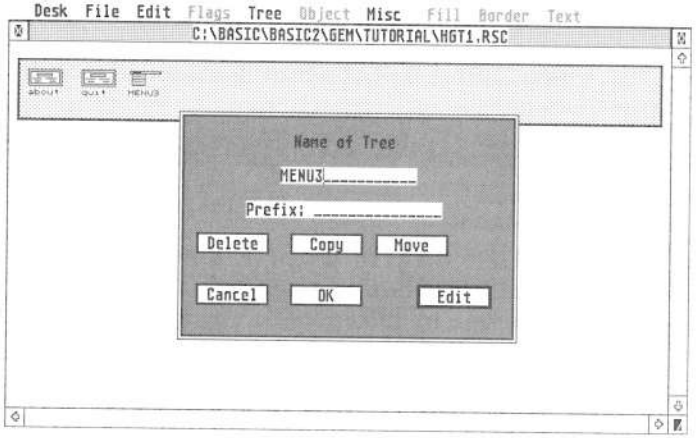
The finished Quit box should look like this:




*The completed Quit box*

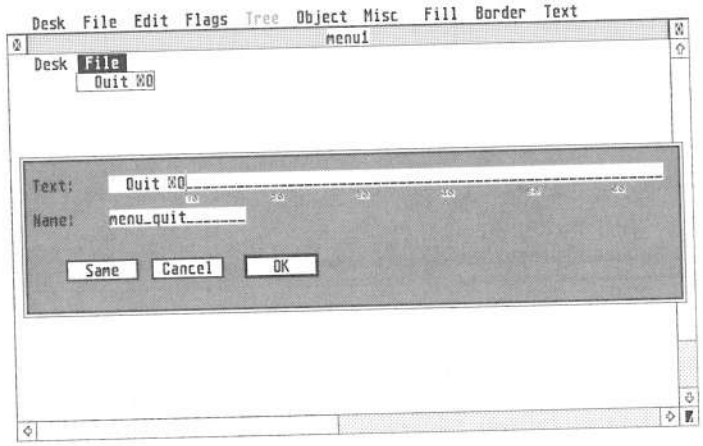
and you should use the names `qu_text`, `qu_ok` and `qu_cancel` for the text and two buttons respectively. You can make the Cancel button a default button (selected when the user hits Return) by selecting it while editing the dialog box and then choosing Default on the Flags menu.

Now all we need is a menu to allow us to select these two dialog boxes, together with some keyboard shortcuts for them. So, select Menu from the Tree menu:




*Creating a menu*

Press Escape and name our menu `menu1`, then click Edit (or press Return). You will see a prototype menu appear with the menu titles Desk and File, as defaults. We will keep these titles. Click on the menu title File and you will see the menu entry Quit appear below it; again, this is a default provided for you by WERCS. Now double-click on Quit to edit this menu item.

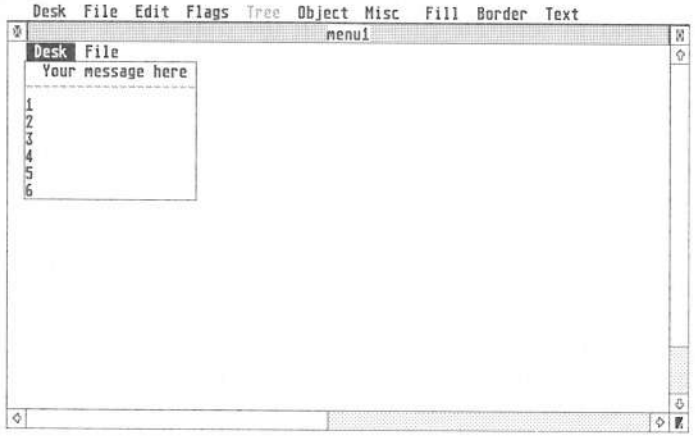



*The Quit item with a keyboard shortcut*

The default text (`Quit`) seems appropriate for this menu item so we just need to add a keyboard shortcut; type a space followed by Ctrl-G and a capital Q which will add the keyboard shortcut Alt-Q to this menu item. Press Escape and name the item `menu_quit` and click OK.



Now click on the Desk menu title:




*Creating the About menu item*

We are going to have an About menu item at the top of the Desk menu with a keyboard shortcut of Alt-A, so double-click on the Your message here line and, when the box appears, hit Escape and type two spaces, `About`, one space, Ctrl-G and capital A; then hit Tab and name this item `menu_about` and click OK.

The menu and the two dialog boxes are finished; click the *Close* box in the top left-hand corner to get back to WERCS' root screen and then click the *Close* box again to exit the resource editor; you will be asked to confirm that you want the changes that you have made saved - click Save..

You will now be returned to the HiSoft BASIC editor.

## Writing the program structure

What does this program do?

Open a window, with a menu bar; fill the window with text; allow the user to use the window gadgets (like the scroll bar etc.) and respond to any menu selections in an appropriate way - Quit to leave the program with a message, About to give a brief message about the program.

Since we are going to use the *HiSoft GEM Toolbox* to perform most of the work for us, it would be as well to understand how to use the toolbox before proceeding much further.

HGT is a collection of constants, sub-programs and functions that are supplied in a modular form, with routines common to particular concepts held in separate files e.g. all the dialog box routines can be found in DIALOG.BAS, the menu routines are in MENU.BAS etc. If you are using the toolbox in its modular form, you will also need to use TOOLBOX.BAS which contains most of the routines that are needed by all the other parts of HGT. We have supplied as HGT.BAS the entire toolbox; this file just includes all the individual components.

In order to use the routines in your program you can incorporate the necessary sub-programs and functions directly in your program or you can 'include' the relevant source files at compilation time (using `REM $include filename`) or you can pre-compile the required toolbox modules into a pre-tokenised .T file. The latter method gives you the fastest development time since it minimises the time taken to compile your program so this is the method we will use; the only problem is that we need to know which toolbox modules we are going to use and then we need to pre-compile them.

Our program opens a text window so we need WINDOW.BAS & TEXTWIND.BAS and uses dialog boxes & menus, thus requiring DIALOG.BAS & MENU.BAS. Also we will have to use TOOLBOX.BAS since, as we mentioned above, this has initialisation procedures and constants for the library.

> Note that the GEMAES.BH file is included by TOOLBOX.BAS, so there is no need to include it in your own program if you use the toolbox.

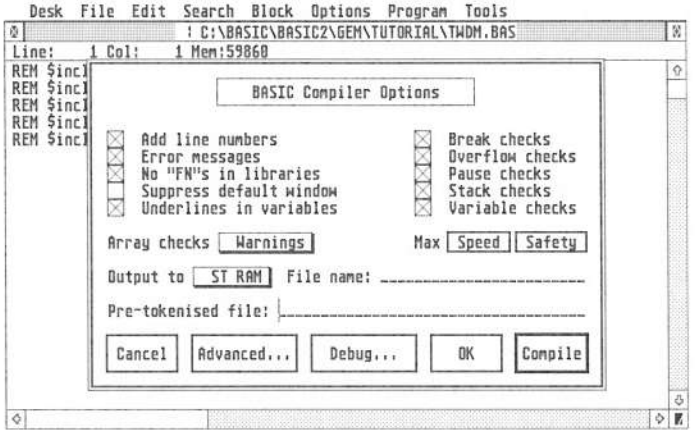
So let's pre-compile this collection of the toolbox modules: type the following into a new HiSoft BASIC window

```
REM $include TOOLBOX.BAS
REM $include WINDOW.BAS
REM $include TEXTWIND.BAS
REM $include DIALOG.BAS
REM $include MENU.BAS
```

You do not need to use upper-case for the filenames but it aids the readability of the program. Save the program as TWDM.BAS (this stands for Text, Windows, Dialogs, Menus) - the editor will put this in the current directory path which should be your TUTORIAL folder.



Now select Compile... from the Program menu, or press Alt-C and check that your Compile box looks like this:




*The Compile box in preparation for pre-tokenising*

We've left all the compile checks on because we are developing the program and want full error reporting. You may want to turn them off when you have completed the program to obtain maximum speed. Hit OK (not Compile). Before we dump the tokens we need to ensure that the compiler will be able to find the toolbox files. You should check that the INCBAS environment variable (set by selecting Environment... on the Tools menu) is set to the directory where you have the toolbox source. Now we can choose Dump tokens from the Program menu (Alt-D).

After some time the pre-tokenising will have finished and you will have a file called TWDM.T in your TUTORIAL folder - this is the pre-tokenised file that we will use to access all of the toolbox that we need in our program. Close the window and select New from the Edit window to begin creating our first HGT program.

To tell the compiler to use this pre-tokenised file, select Compile... from the Program menu and type `TWDM.T`; this will be entered next to Pre-tokenised file: at the bottom of the box; click OK (not Compile).

> The compiler will look for the pre-tokenised file first in the path specified by the INCBAS environment variable (set by selecting Environment... on the Tools menu) and then in the current directory.

Now for the program itself ...

Remember we are opening a window with a menu, loading text into it and responding to use of the menu bar. First things first; whenever we are going to use the toolbox for menus, windows or resource files, we will need to call StartProgram to initialise various structures. Basically, StartProgram loads a resource file for you and uses the menu that you give as a parameter, it also handles the exit from the menu structure automatically or allows you to process Quit yourself.

Every menu should have a Quit item (normally on the File menu) and, to get HGT to quit automatically for you when this item is selected, you only have to pass the item's name as a parameter to StartProgram. If, instead, you specify -1 as the exit item you will have to handle the Quit code yourself - in our finished program this is what we will do because it is good practice and allows more flexibility (i.e. you can ask for confirmation). However to start with, for simplicity, we will let the toolbox handle Quit for us. Let's write the outline of our program:

```
' Your first HGT program
' Displays a window with a menu
REM $option y, l10

'Get the resource file header
REM $include HGT1.BH

StartProgram "HGT1.RSC",menu1,menu_quit

HGTLoop

StopProgram
```

There are a few things that we haven't talked about here:

```
REM $option y, l10
```

This uses options y and l10 to suppress the automatic GEM window that HiSoft BASIC gives you and to leave 10Kb for resource file usage respectively. We don't want the automatic window because our code doesn't open a window yet and the automatic window would therefore be 'at the front' which means that the toolbox would not be able to respond to our menus. As regards the 'leave' option this should allocate memory of roughly the size of the resource file - we have been over-generous with 10Kb.



```
REM $include HGT1.BH
```

This includes the resource file's BASIC header file which holds all the names of the object used so that we can refer to them in our program.

```
StartProgram "HGT1.RSC",menu1,menu_quit
```

Initialise the toolbox, load the resource file HGT1.RSC, use the menu tree given by menu1 within that resource file and use menu_quit as the menu item that quits the running program.

```
HGTLoop
```

You must always call this since it is the controlling heart of the toolbox. It will respond to mouse clicks, keyboard entries and so on, passing control to its own or your own routines as appropriate.

```
StopProgram
```

Another mandatory toolbox routine that frees up all the resources that your program has used.

Type in this simple program and run it (Alt-X); it will take a little time to compile because we are using most of the toolbox. When it is running move the mouse up to the menus and you should see an About item on the Desk menu and a Quit item on the File menu - don't select About since we have no code to handle that yet, instead select Quit. Hopefully, you will be returned to the editor.

Try running the program again and typing Alt-Q - this should also Quit the program since we defined it as a keyboard shortcut.

If you have problems, check your code carefully. If you lose your menus you will have to exit the editor using Alt-Q and reload HiSoft BASIC and your program - it is easy to confuse GEM.

In order to extend our program to handle the menu items, we must include a routine called ProcessUserMenus since it is this routine that HGT will call when a user selects a menu item. You will always need such a routine when you write a program with menus.

This just about completes the outline structure and the program looks like this:

```
Desk  File  Edit  Search  Block  Options  Program  Tools
                : C:\BASIC\BASIC2\GEM\TUTORIAL\HGT1.BAS
Line:   11 Col:    8 Mem:59711
' Your first HGT program
' Displays a window with a text file and menu

REM $option y,l10

' Get the resource file header
REM $INCLUDE HGT1.BH

SUB ProcessUserMenus (VAL wind, VAL item, VAL title)

END SUB

StartProgram "HGT1.RSC",menu1,menu_quit

HGTLoop

StopProgram
```

*The outline structure of your first program*

We haven't written the `ProcessUserMenus` sub-program yet, we've just included its variable definitions so that the program will compile.

## Completing the program

Now we must write the code that will fill the window with text and then code the `ProcessUserMenus` sub-program. Let's fill the window first since this is a little easier and less prone to error than handling the menus.

Remember that this is going to be a text window so we must first initialise the relevant variables using the toolbox routine `InitTextWindows` which takes two parameters; the first is the number of text windows that you are going to use in your program (in this case, 1) and the second parameter is the maximum number of lines any one window will hold (let's say 200 for our program). The call to this sub-program also sets up the `TextWindows` variable which will be used when loading text into the window - it is the number of the next available window. So we must save this value of `TextWindows` (in our own variable `MyWindow`, say) before opening the window.



Now we can call the function `OpenTextWindow` which opens a window, gives it a title and allows you to specify which window gadgets you want used; we will use the pre-defined constants `win_all` and `win_info` to say that we want all the gadgets except the information bar at the top.

The call to `OpenTextWindow` returns the window's AES handle.

Text is displayed from within a string array called `lines$` which has two dimensions, the first is the window number and the second is the actual lines of text; `lines$` is dimensioned by the toolbox when you call `InitTextWindows`. To load text into this array from a file we can use the toolbox sub-program called `LoadTextFile` which needs the name of the file and the window number (given by `MyWindow`) as its first two parameters. The third parameter is a variable parameter and is used to return the number of lines read in; this should be set up in a toolbox array variable `wolines()` so that the window handler knows how to set up the vertical scroll bar etc.

After that lengthy pre-amble, here's the code to set up a text window and read a file into it:

```
Desk  File  Edit  Search  Block  Options  Program  Tools
                : C:\BASIC2\HGT\TUTORIAL\HGT1.BAS
Line:   25 Col:    1 Mem:59522
' Get the resource file header
REM $include HGT1.BH

CONST maxlines=200

SUB ProcessUserMenus(VAL wind, VAL item, VAL title)

END SUB

StartProgram "HGT1.RSC",menu1,menu_quit

InitTextWindows 1,maxlines
MyWindow=TextWindows
whandle=OpenTextWindow("Your first GEM program",win_all-win_info)
LoadTextFile "HGT1.BAS", MyWindow, wolines(MyWindow)

HGTLoop

StopProgram
```

*Opening a text window*

Note that we have declared a constant (`maxlines`) for the maximum number of lines in our text window.

Try running this program - assuming that it compiles and runs you should be able to choose Quit from the File menu or type Alt-Q to return to the editor. Do not select About since we have no code to handle this yet.

Finally we have to write the ProcessUserMenus sub-program. As we can see from the shell of this routine, it takes three parameters; all of these are passed by value and they give the AES handle of the top window so that you can ensure that the correct window is being processed, the item that has been selected from the window's menu and the title of the menu selected.

This is where the .BH file comes in that was created when we saved our resource file. In this file are BASIC constants that associate the names of the objects that you defined using WERCS with the object numbers within the tree structure that is the resource file.

> The .HRD file, that was also created when you saved the resource file, contains the names of the objects and trees as well, but in a generic WERCS format so that the resource editor can name all your objects as you intended when you load the resource file for editing. Without the .HRD file WERCS still functions but will invent names for objects.

So, you can simply use these names within your BASIC program to refer to the objects in the tree. For example, remembering that we called the About item on the Desk menu menu_about, then as long as we include the .BH file we can write something like:

```
SUB ProcessUserMenus (VAL wind, VAL item, VAL title)
SELECT CASE item
  CASE menu_about
```

and whatever follows this CASE selection will be obeyed when the About box is selected. Now, what we would like to do when the About item is selected is to bring up the About dialog box that we called about when we created it in WERCS - we do this by selecting the object tree which is the dialog box and then calling a toolbox routine called HandleDialog which will return control when the user has clicked something in the dialog box:

```
  CASE menu_about
    SelectTree about
    dum=HandleDialog(0)
```



This will display the About dialog when the Alert item is selected and let the user click OK, when it will return with the object number that was selected in dum.

Well, how about Quit? In this case there are two buttons to click so we have a little more work to do. Consider:

```
  CASE menu_quit
    SelectTree quit
    dum=HandleDialog(0)
    IF dum=qu_ok THEN Finished_Flag=1 ELSE Finished_Flag=0
```

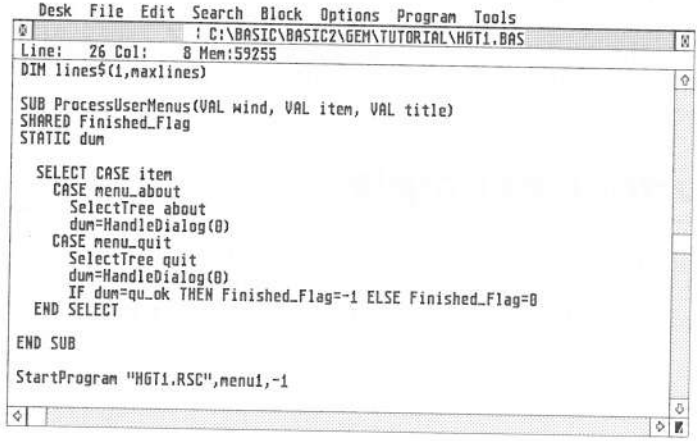
So, if the user clicks OK, Finished_Flag will be set to 1; otherwise, if the user clicks Cancel (or hits Return since this is the default button), Finished_Flag will be set to 0.

What is Finished_Flag? It's a toolbox variable that lets HGTLoop know that we have decided to quit the program. Remember that when we issued a StartProgram we set the last parameter in this sub-program call to menu_quit so that this would be the exit item. Well, now that we have handled Quit ourselves and set up Finished_Flag appropriately, we can change our call to StartProgram to look like:

```
StartProgram "HGT1.RSC",menu1,-1
```

The -1 means 'the program will do its own Quit handling'.

The finished ProcessUserMenus looks like this:




*The ProcessUserMenus sub-program*

We've now completed the program. Save it and then try running it.

## Testing and Debugging

Run the program and, assuming that you have typed it in correctly, it should work first time. Try selecting About both by choosing it from the menu and by pressing Alt-A. Also make sure that you can cancel a Quit. Use the scroll bars on the window and try the *Full* box and the *Grow* box. Finally Quit the program.

Hopefully, everything works fine - if not carefully check your program against the listings above and re-compile.

It may have occurred to you that the OK button in the About box should be a default button. Let's do that.

Save your source code and select WERCS from the Tools menu; the resource file should be loaded automatically for you. When it appears, double-click on the about object, select the OK button (click on it so that it goes black) and then select Default on the Flags menu.

Now all you have to do is Quit WERCS (the resource file will be saved for you) and re-compile your program. Run it, and you should be able to press Return to exit the About box. Simple, isn't it?

That completes your first HGT program. Before you leave it, you might like to try to write another clause to the CASE statement in ProcessUserMenus that handles the loading of the text file into the window via a Load item on the menu bar. The solution is given at the end of this chapter. *Hint*: you will need to use FullRedraw whandle after loading the text to re-draw the window.

# A further example

We will now look at a more complex example but in less detail. One of the example programs that is supplied with HiSoft BASIC is called SHELL.BAS, here is a listing (on the next page):



```
REM $option k150,y
REM $include shell.bh

SUB ProcessUserMenus(VAL cur_front,VAL item,VAL title)
SHARED texth,wolines(1),lines$(2),TextWindows,ImageWindows
STATIC i,fi$,temp,id,curline$,Comm

SELECT CASE item
  CASE MeAboutShell: junk=NewForm_Alert(AboutAlert,1)
  CASE MeShowTextFi:
     fi$=FileSelect$
     IF fi$<>"" THEN
       LoadTextFile fi$,TextWindows,wolines(TextWindows)
       texth=OpenTextWindow(fi$,win_all-win_info)
     END IF

CASE MeShowImageF:
     fi$=FileSelect$
     IF fi$<>"" THEN
       LoadPicture fi$,imagewindows
       texth=OpenImageWindow(fi$,win_all-win_info)
     END IF

CASE MeRunProgram:
     fi$=FileSelect$
     IF fi$<>"" THEN Execute fi$,curline$

CASE MeSetCommandL:
     SelectTree CLine
     Sette_ptext ClCommandLine,curline$
     Comm=HandleDialog(CLCommandLine)
     SELECT CASE Comm
       CASE ClOK:
          curline$=Gette_ptext$(CLCommandLine)
       END SELECT
END SELECT

END SUB

StartProgram "SHELL.RSC",MENU1,MEQuit
InitTextWindows 2,1500
DIM images(2,fd_size)
expandtabsflag=-1
HGTLoop
StopProgram
```

This example uses a large proportion of the toolbox routines and needs to be compiled with the pre-tokenised file ~~MOST.T~~ HGT.T included in the HGTTUTOR folder .

Let's look at what it does:

## The main loop

The two REM statements at the beginning allocate 150Kb for the BASIC heap leaving the rest to the operating system (`$option k150`), turn off the default GEM window that HiSoft BASIC gives you (`y`) and include the names of the GEM objects that will be used by the program (`$include shell.bh`).

Of course, there is a `ProcessUserMenus` sub-program because this example is menu-driven and we will come to this shortly; we'll look at the main program statements first:

`StartProgram` initialises the toolbox, defines the resource file to be used (`SHELL.RSC`), selects the menu to be displayed (`MENU1`) and the item on that menu that is to be the exit item (`MEQuit`).

`InitTextWindows`, as we have seen in the first example, sets up the text window handling, in this case allowing a maximum of two text windows with no more than 1500 lines in each. It is the program's responsibility to ensure that no more than two text windows are opened and that no more than 1500 lines are placed in a window - the error checking routines are missing from the above example (perhaps you would like to add them?).

The `DIM` statement declares the `images` array which is used by the toolbox to hold any images that you may wish to display using `OpenImageWindow`. As you can see, the program allows for 2 image files to be displayed, although it does no error checking. `fd_size` is a constant defined within TOOLBOX.BAS.

`expandtabsflag` is an HGT variable which, if set (non-zero), will expand tabs to a number of spaces; how many spaces is determined by the `tabsize` variable, whose default value is 8.

Finally we have `HGTLoop` which will handle all user interaction via the `ProcessUserMenus` sub-program and, of course, `StopProgram` to terminate the program cleanly.



## The ProcessUserMenus sub-program

This sub-program is the heart of SHELL.BAS; it responds to all the menu selections made by the user. The code begins by declaring some variables as `SHARED` - these are the toolbox variables that control the handling of text and image windows. Some variables that are local to `ProcessUserMenus` are then declared.

The main code of the sub-program is just a series of `CASE` statements which respond to the menu items that have been selected by the user. The names (`MeAboutShell`, `MeShowTextFi` etc.) are the names given to the strings of the menu items as created by WERCS.

`MeAboutShell` displays the About alert box by calling `NewForm_Alert` with the name of the alert object created in WERCS and the default button number. Buttons within alert boxes cannot be named, hence the use of a button *number* in this routine.

`MeShowTextFi` uses the `FileSelect$` function to display the file selector and prompt for a filename The pathname is returned (or a null string if the user clicked Cancel) so this code then checks to see if a pathname has been returned and, if it has, it loads the file specified into the `lines$` array at the current index (given by `TextWindows`) and then opens a text window with the pathname as the title.

`MeShowImagF` does a very similar task to `MeShowTextFi`; a pathname is obtained, checked if non-null and then the file is loaded into the `images` array at the current index (given by the `imagewindows` variable maintained by the toolbox). Finally, the image is displayed in an image window.

Neither of these window routines check to see if the file is too long or if too many windows have been opened - the whole program needs work to ensure that it is 'user-proof'.

`MeRunProgram` simply executes a selected program by calling `Execute` with a command line that has been set up in `curline$` (see below).

`MeSetCommandL` is used to set up the command line for `MeRunProgram`. Firstly, it selects the `Cline` dialog box (run WERCS and load the resource file SHELL.RSC to see the definition of this dialog box) and then copies text from the `curline$` local variable into the `ClCommandLine` object within this dialog box. `HandleDialog` is then called and the cursor placed within the `CLCommandLine` field, ready for editing.

Assuming that the user presses OK to exit the dialog box and therefore `HandleDialog` returns `ClOK` (the name of the OK button) the code then assigns whatever text has been entered into the `CLCommandLine` field into the `curline$` variable, using the `Gette_ptext$` function.

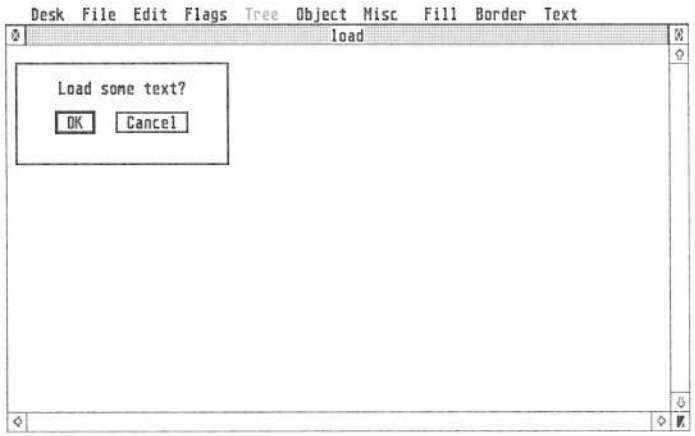
So that's all there is to it. A simple and very modular program that nevertheless does a great deal of work for you. Of course, it would benefit from some error checking ...!



# Solution to Exercise

We asked you to add a Load item to the menu that would allow the user to choose to load a text file. The first step in doing this is to use WERCS to add a menu entry and a dialog box. The dialog box is not necessary but adds some style.

Enter WERCS from HiSoft BASIC and, assuming you were editing HGT1.BAS in BASIC, the resource file will be loaded for you; otherwise load the resource file yourself. Design a new dialog box (named `load`) that looks like:

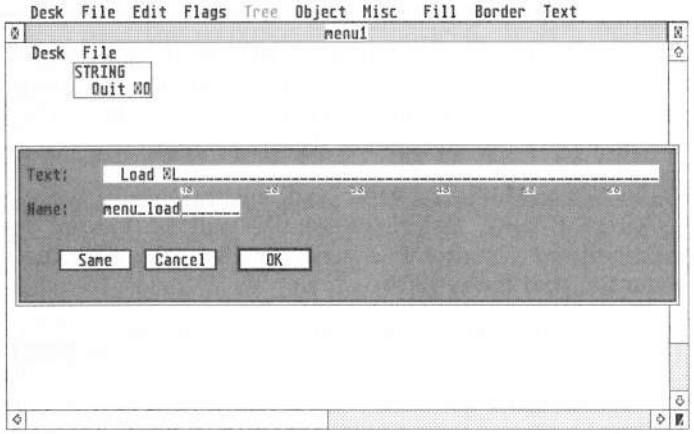



*The Load dialog box*

Name the OK button `load_ok` and the Cancel button `load_cancel`. Return to the root screen of WERCS and double-click on the menu1 tree to edit it.

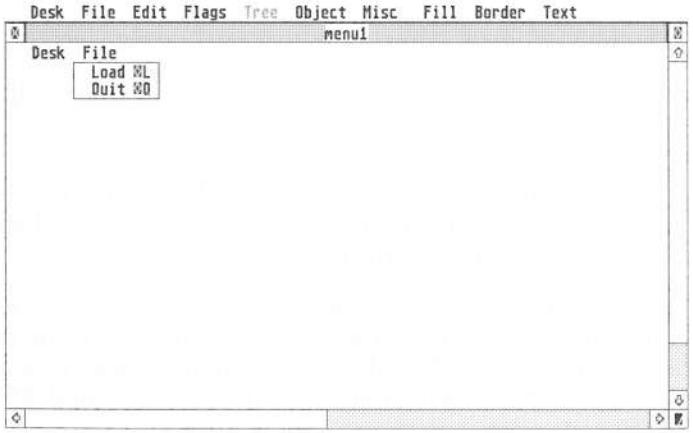
Add a new menu item onto the File menu; click on File to bring up the menu, choose String from the Object menu, place the pointer *over the top and to the left* of the Quit item and click. This adds a correctly aligned new menu item into the menu, called STRING, above Quit.

Double-click on STRING to edit it:




*Adding a new menu item*

The text of the new menu item should be Load (with two preceding spaces); add a keyboard shortcut of Alt-L and call it menu_load. Then click OK and ensure that the menu list is correctly aligned like this:




*The Load item on the File menu*

Quit WERCS, save the file and write the code to handle this new menu item.

The program extended to handle loading a text file into the window from a Load item on the menu is shown on the next page.



```
' Your first HGT program
' Displays a window with a text file and menu

REM $option y,l10

' Get the resource file header
REM $include HGT1.BH

CONST maxlines=200

SUB ProcessUserMenus(VAL wind, VAL item, VAL title)
SHARED Finished_Flag,MyWindow,wolines(1),whandle
STATIC dum

  SELECT CASE item
    CASE menu_about
      SelectTree about
      dum=HandleDialog(0)
    CASE menu_quit
      SelectTree quit
      dum=HandleDialog(0)
      IF dum=qu_ok THEN Finished_Flag=-1 ELSE
Finished_Flag=0
    CASE menu_load
      SelectTree load
      dum=HandleDialog(0)
      IF dum=load_ok THEN
        LoadTextFile "HGT1.BAS",MyWindow,wolines(MyWindow)
        FullRedraw whandle
      END IF

  END SELECT

END SUB

StartProgram "HGT1.RSC",menu1,-1

InitTextWindows 1,maxlines
MyWindow=TextWindows
whandle=OpenTextWindow("Your first GEM program",win_all-
win_info)

HGTLoop

StopProgram
```

## End Sub

That was a tour of the concepts that underly HiSoft BASIC and the *HiSoft GEM Toolbox*. Hopefully it has provided you with a good enough grounding to begin to experiment with the various commands in detail.

There are several ways to expand your BASIC programming skills.

Firstly practice. You will get a feel for how the different commands work by trying them out in routines of your own devising.

Secondly type in routines from magazines and books (you will be pleasantly surprised by how many listings work in HiSoft BASIC with only the minimum of modification so you do not even have to restrict yourself to Atari programs). There is never one correct way of programming a given task, any dozen people would probably come up with a dozen different solutions to a given problem. Entering listings is therefore an invaluable way of learning new ways of looking at things.

Remember, you do not have to re-invent the wheel every time you sit down at the keyboard. Many routines have already been created that can be used within your own programs, probably with greater speed or using less memory space than you would have been able to come up with on your own.

The bibliography lists a selection of books that may be useful in teaching programming techniques.



# Chapter 3
# HiSoft BASIC 2
# The Program

## Introduction

HiSoft BASIC 2 (called simply HiSoft BASIC from now on) is a complete package for the production of fast, efficient programs on your Atari ST/STE/TT computer.

There is an *editor* for the creation and editing of your BASIC source code, a *resource editor* for creating GEM objects such as menus and dialog boxes for use with BASIC, a *debugger* for helping you to stamp out those nasty bugs (problems) in your programs and, of course, a *compiler* to turn your BASIC source code into speedy, compact machine code. We also supply some other tools to make your life easier including a *RAM disk* and a *profiler*, which analyses your BASIC program and shows you where it can be made to run more quickly.

To take the best advantage of the Atari computers without incurring a high memory overhead, HiSoft BASIC uses library functions and sub-programs to access most of the operating system. There are libraries to handle all aspects of the different Atari systems including a high level GEM toolbox which unravels many of the mysteries hitherto surrounding this part of the operating system.

We have also provided a host of example files on the disk, many of which will have been referred to in the previous chapter.

This chapter looks at using the various tools that are supplied with HiSoft BASIC and aims to give you a friendly introduction to all aspects of the package - it does not detail the language itself or the technical aspects of the software; this is covered in the *Technical Reference* manual.

# The Editor

The editor supplied with HiSoft BASIC is fully integrated with the system which means that you can develop programs in an intuitive and interactive manner, creating and editing your programs in the same environment as running and debugging your finished masterpiece.

Moreover, those of you with strong preferences for your own editor can dispense with the HiSoft editor and use your own favourite package along with the TTP version of HiSoft BASIC; although you will lose the benefits of interactive development.

The editor for HiSoft BASIC is a multi-window screen editor which allows you to enter and edit text and save and load from disk, as you would expect. It also lets you print some or all of your text, search and replace text patterns and use any of your computer's desk-accessories. It is GEM-based, which means it uses all the user-friendly features of GEM programs that you have become familiar with such as windows, menus and mice. However, if you're a die-hard used to the hostile world of computers before the advent of WIMPs, you'll be pleased to know you can do practically everything you'll want to do from the keyboard without having to touch a mouse.

The editor is 'RAM-based', which means that the file you are editing stays in memory for the whole time, so you don't have to wait while your disk grinds away loading different sections of the file as you edit. As the ST/TT range of computers has so much memory, the size limitations often found in older computer editors do not exist with HiSoft BASIC. As all editing operations, including things like searching, are RAM-based they act extremely quickly.

When you have typed in your programs it is not much use if you are unable to save them to disk, so the editor has a comprehensive range of save and load options, allowing you to save all or part of the text and to load other files into the middle of the current one, for example.



To get things to happen in the editor, there are various methods available to you. Features may be accessed in one or more of the following ways:

- Using a single key, such as a Function or cursor key;
- Clicking on a menu item, such as Save;
- Using a menu shortcut, by pressing the Alternate key (subsequently referred to as Alt) in conjunction with another, such as Alt-F for *Find*;
- Using the Control key (subsequently referred to as Ctrl) in conjunction with another, such as Ctrl-A for *cursor word left*;
- Clicking on the screen, such as in a scroll bar.

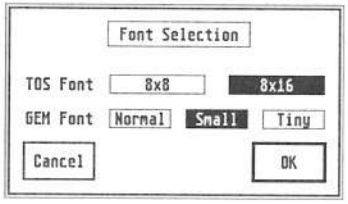
The menu shortcuts have been chosen to be, hopefully, easy to remember.

# A word about pop-up menus and dialogs

The editor makes extensive use of dialog boxes and pop-up menus, so it is worth recalling how to use them, particularly for entering text. The editor's dialog boxes contain buttons, radio buttons, and editable text.

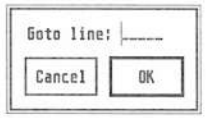
Exit buttons may be clicked on with the mouse and cause the dialog box to go away. Usually there is a default button, shown by having a wider border than the others. Pressing Return on the keyboard is equivalent to clicking on the default button.

Radio buttons are groups of buttons of which only one may be selected at a time - clicking on one automatically de-selects all the others.



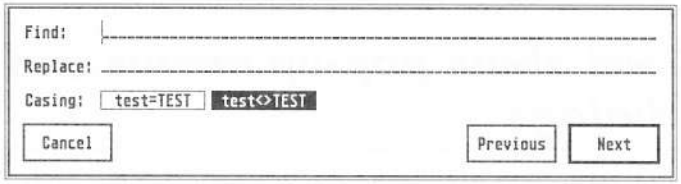

*A dialog with buttons (OK, Cancel) and radio buttons (Normal, Small etc.)*

Editable text is shown with a dotted line, and a vertical bar marks the cursor position.




*Editable text*

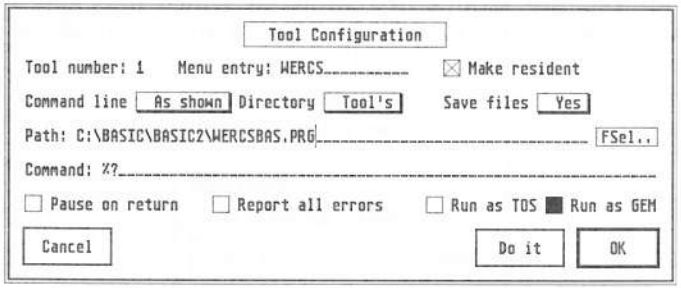
Characters may be typed in and corrected using the Backspace, Delete and cursor keys. You can clear the whole edit field by pressing the Esc key. If there is more than one editable text field in a dialog box, you can move between them using the Tab key or the ↓ and ↑ keys or by clicking near them with the mouse.




*More than one editable text field*

Some dialog boxes allow only a limited range of characters to be typed into them - for example the Goto... dialog box only allows numeric characters (digits) to be entered.

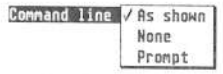
As well as the conventional GEM user interface facilities, the editor also uses some extensions. To illustrate these, consider the dialog box shown below:




*The Tool Configuration dialog box*



Some options are accessed via 'pop-up' menus similar to those used by Atari's new control panel. Thus if you move the mouse over the As shown selection (by Command line) and press down on the left mouse button, a menu like this will pop up:



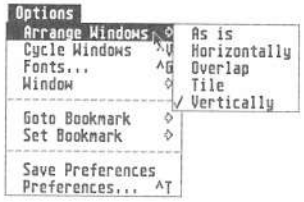

*A pop-up menu*

This indicates that the current setting for this option is As shown. The mouse will highlight the current selection that you are making and when you let go of the mouse this indicates that you have made your selection. If you let go outside the pop-up menu then this is taken as cancelling the selection.

The box beside Make resident has a cross in it, indicating that this option is selected; similarly Report all errors is *not* selected. Clicking in one of these boxes, or the associated text, will cause that option to be toggled on and off.

Run as TOS and Run as GEM are a pair of 'radio options'; the solid box indicates the currently selected item: clicking on Run as TOS will change both boxes.

Some of the menu items on the main 'drop-down' menus now have sub-menus; these are indicated by a ⇨ symbol. For example:




*A sub menu*

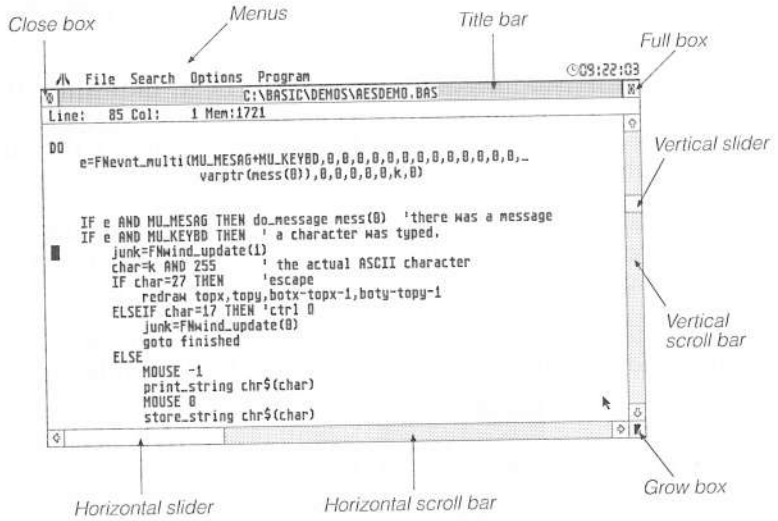
When you highlight a menu item (like Arrange Windows in the example above), the corresponding sub-menu will appear after a short delay. You can then move the mouse to the right to select the particular item that you want. To cancel the operation just let go of the mouse without selecting a sub-item or move to another item from the main menu.

If the editor doesn't have enough room to display the sub-menu to the right of the main menu, it will do so on the left; the items are selected in the same way.

# The Editor's windows

Having loaded HiSoft BASIC, you will be presented with an empty window with a status line at the top and a flashing black block, which is the *cursor*, in the top left-hand corner.

The window used by the editor works like all other GEM windows, so you can move it around by using the *Title* bar on the top of it, you can change its size by dragging on the *Grow* box, and make it full size (and back again) by clicking on the *Full* box. Clicking on the *Close* box is equivalent to choosing Quit from the File menu.




*A GEM window*

The status line contains information about the cursor position in the form of Line and Column offsets as well as the number of bytes of memory which are free to store your text. Initially this is displayed as 9980, as the default text size is 10000 bytes. You may change this default if you wish, together with various other options, by selecting Preferences, described later. The 'missing' 20 bytes are used by the editor for internal information. The rest of the status line area is used for error messages, which will usually be accompanied by a 'ping' noise to alert you. Any message that is printed will be removed when subsequently you press a key.



## Switching Windows

The editor has support for up to six windows, which can be selected by pressing Alt-1 to Alt-6 (on the top row of numbers, *not* on the numeric pad). The windows can be organised in a number of ways and you can select this using Arrange Windows on the Options menu. Try this out for yourself to get the idea of how the different arrangements work.

You can cycle through the open windows using the Cycle Windows command from the Options menu (or use Ctrl-V), by clicking on the appropriate window with the mouse or by selecting the appropriate sub-item from the Window item on the Options menu.

To close a window and thus free the memory used by it, click on its close box or use the Ctrl-W key combination.

To cut and paste between windows is just as simple as copying blocks in a single window, i.e. mark the block and then use the Cut command, switch windows (as described above) and then Paste. See below for more detail on cut and paste.

# Entering text and moving the cursor

To enter text, simply type on the keyboard and at the end of each line press the Return key (or the Enter key on the numeric pad) to start the next line. You can correct your mistakes by pressing the Backspace key, which deletes the character to the left of the cursor, or the Delete key, which removes the character on the cursor.

## Cursor keys

To move the cursor around the text to correct errors or enter new characters, you can use the cursor keys, labelled ← → ↑ and ↓ or the mouse; move the cursor to a specific position on the screen with the mouse pointer and click. If you position the cursor past the right-hand end of the line and type some text at that point the editor will automatically add the text to the real end of the line. If you type in long lines the window display will scroll sideways if required.

When you cursor up at the top of a window the display will either scroll down if there is a previous line, or print the message `Top of file` in the status line. Similarly if you cursor down off the bottom of the window the display will either scroll up if there is a following line, or print the message `End of file`.

You can move the cursor on a character basis by clicking on the arrow boxes at the end of the horizontal and vertical scroll bars.

To move immediately to the start of the current line, press Ctrl ←, and to move to the end of the current line press Ctrl →.

To move the cursor a word to the left, press Shift ← and to move a word to the right press Shift →. You cannot move past the end of a line with Shift →. A word is defined as anything surrounded by a space, a tab or a start or end of line. The keys Ctrl-A and Ctrl-F also move the cursor left and right on a word basis.

To move the cursor a page up, you can click on the upper grey part of the vertical scroll bar, or press Shift ↑. To move the cursor a page down, you can click on the lower grey part of the scroll bar, or press Shift ↓.

## Tab key

The Tab key inserts a special character (ASCII code 9) into the buffer, which on the screen looks like a number of spaces, but is rather different. Pressing Tab aligns the cursor onto the next 'multiple of 4' column, so if you press it at the start of a line (column 1) the cursor moves to the next multiple of 4, +1, which is column 5. Tabs are very useful indeed for making items line up vertically and its main use in HiSoft BASIC is for such things as indenting structured program lines. When you delete a tab the line closes up as if a number of spaces had been removed. The advantage of tabs is that they take up only 1 byte of memory, but can show on screen as many more.

You can change the tab size before or after loading HiSoft BASIC; to change the default use the Preferences command described shortly.



## Backspace key

The Backspace key removes the character to the left of the cursor. If you backspace at the very beginning of a line it will remove the 'invisible' carriage return and join the line to the end of the previous line. Backspacing when the cursor is past the end of the line will delete the last character on the line, unless the line is empty in which case it will re-position the cursor on the left of the screen.

## Delete key

The Delete key removes the character under the cursor and has no effect if the cursor is past the end of the current line.

## The Edit menu


The commands on the top of the Edit menu may be used to perform the conventional Cut, Copy and Paste operations on marked blocks.

These are described under *Block commands*, below.

## Goto line

To move the cursor to a specific line in the text, click on Goto... from the Edit menu, or press Alt-G. A dialog box will appear, allowing you to enter the required line number. Press Return or click on the OK button to go to the line or click on Cancel to abort the operation. After clicking on OK the cursor will move to the specified line, re-displaying if necessary, or give the error End of file if the line doesn't exist.

Another fast way of moving around the file is by dragging the slider on the vertical scroll bar, which works in the usual GEM fashion.

## Go to top of file

To move to the top of the text, click on Goto Top from the Edit menu, or press Alt-T. The screen will be re-drawn if necessary starting from line 1.

## Go to end of file

To move the cursor to the start of the very last line of the text, click on Goto Bottom, or press Alt-B.

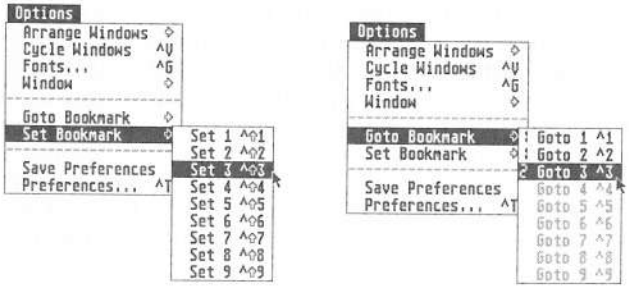
## Show Tokens

When chosen, this will upper-case all the HiSoft BASIC reserved words in the current window. This is very useful as a quick syntax check since you can see immediately which words you have mis-typed. It also makes your programs easier to read and to debug.

See also Show BASIC Tokens under Preferences below.

# Bookmarks

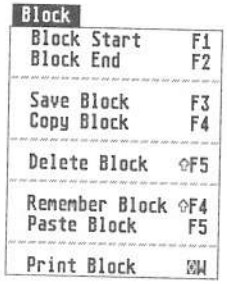
A further way to navigate your source text is via the use of bookmarks. A bookmark is set by selecting the appropriate Set Bookmark item from the Options menu or by using Ctrl-Shift and a digit key (not the numeric keypad). When you set a bookmark the corresponding item on the Goto Bookmark menu will become enabled. Then, selecting this item, or by pressing Ctrl and the digit, will return you to the original position.






When you set a bookmark, the window number to which it refers is displayed in the menu. Going to a bookmark may cause you to switch windows. Note that bookmarks that are set in a given window are lost when you close that window.

# Block Commands




A *block* is a marked section of text which may be copied to another section, deleted, printed or saved onto disk. Blocks may be marked using the mouse, via menu items or with function keys.

A marked block is highlighted by showing the text in reverse. While you are editing a line that is within a block this highlighting will not be shown but will be re-displayed when you leave that line or choose a command.

## Marking a block

The simplest way to mark a block is to click on the first character in the block and drag the mouse to the end of the block. The block will be highlighted by showing the text in reverse as you drag the mouse. When you move the mouse to the bottom of the window, the window will scroll. Conversely, moving the mouse to the top of the window, will cause the window to scroll in the opposite direction. You may start marking a block, by clicking at the end if you wish.

Double-clicking will cause the word 'under' the mouse to be marked as the block. If you double-click and then drag, text will be highlighted a word at a time. Clicking in the the left hand margin of the window causes dragging to occur a line at a time.

The start of a block may also be marked by moving the cursor to the required place and selecting Block Start or pressing key F1. The end of a block can be marked by moving the cursor and selecting Block End or pressing key F2. The start and end of a block do not have to be marked in a specific order - if it is more convenient you may mark the end of the block first.

## The Clipboard: Copy, Cut & Paste

HiSoft BASIC provides conventional clipboard facilities, as popularised by the Apple Macintosh. Once you have marked a block you may copy it to the clipboard by selecting Copy from the Edit menu. The main text will remain as it is. The contents of the clipboard may then be inserted at another position by moving the cursor there and selecting Paste.

The current block may be deleted using Cut from the Edit menu; selecting Paste will then insert the block that was cut (unless you have used Copy in the mean time). Thus to move a block with this method, Cut the block from its original position and then Paste it into its new one.

The block menu also gives you the flexibility of the following commands.

## Saving a block

Once a block has been marked, it can be saved by clicking on Save Block from the Block menu or by pressing key F3. If no block is marked, the message `What blocks!` will appear. If the start of the block is textually after its end the message `Invalid block!` will appear. Both errors abort the command. Assuming a valid block has been marked, the GEM file selector will appear, allowing you to select a suitable disk and filename. If you save the block with a name that already exists the old version will be overwritten - no backups are made with this command.

## Copying a block

A marked block may be copied, memory permitting, to another part of the text by moving the cursor to where you want the block copied and clicking on Copy Block or by pressing key F4. If you try to copy a block into a part of itself, the message `Invalid block!` will appear and the copy will be aborted.



## Deleting a block

A marked block may be deleted from the text by clicking on Delete Block or by pressing Shift-F5. The shift key is deliberately required to prevent it being used accidentally. A deleted block is remembered, memory permitting, in the clipboard, for later use. This is equivalent to Cut on the Edit menu.

## Copy block to block buffer

The current marked block may be copied to the block buffer, memory permitting, using Remember Block or by pressing Shift-F4. This can be very useful for moving blocks of text between different files by loading the first, marking a block, copying it to the block buffer then switching to another window or loading the other file and pasting the block buffer into it. This is equivalent to Copy on the Edit menu.

## Pasting a block

A block in the clipboard may be pasted at the current cursor position by clicking on Paste Block or by pressing F5. This is equivalent to Paste on the Edit menu.


The contents of the clipboard is lost if the edit buffer size is changed and after a compilation.

## Printing a block

A marked block may be sent to the printer by clicking on Print Block or by pressing Alt-W. An alert box will appear confirming the operation and clicking on OK will print the block. The printer port used will depend on the port chosen with the Control Panel, or will default to the parallel port. Tab characters are sent to the printer as a suitable number of spaces, so the net result will normally look better than if you print the file from the Desktop.


If you try to print when no block is marked at all then the whole file will be printed.

Block markers remain during all editing commands, moving where necessary, and are only reset by the commands Delete block and Load.

## Deleting text

### Delete line

The current line can be deleted from the text by pressing Ctrl-Y.

### Delete to end of line

The text from the cursor position to the end of the current line can be deleted by pressing Ctrl-Q

### UnDelete Line

When a line is deleted using either of the above commands it is preserved in an internal buffer, and can be re-inserted into the text by pressing Ctrl-U, or the Undo key. This can be done as many times as required, particularly useful for repeating similar lines or swapping individual lines over.

### Delete block

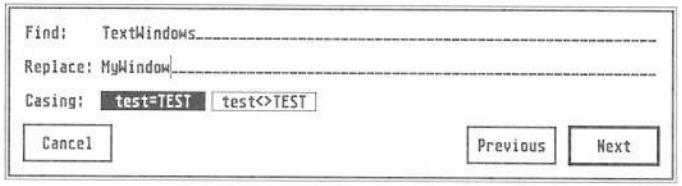
A marked block may be deleted from the text by clicking on Delete Block or by pressing Shift-F5. The shift key is deliberately required to prevent it being used accidentally. A deleted block is remembered, memory permitting, in the clipboard, for later use. This is equivalent to Cut on the Edit menu.



## Searching and Replacing Text


The commands on the Search menu may be used for finding and perhaps replacing existing text. The strings involved are set up by selecting Find or press Alt-F.

This allows you to enter the find and replace strings as shown in the following dialog box:




In the example above TextWindows has been entered as the find string and MyWindow as the replace string.

If you click on Cancel, no action will be taken; if you click Next (or press Return) the search will start forwards, while clicking on Previous will start the search backwards. If you do not wish to replace, leave the replace string empty.

If the search is successful, the screen will be re-drawn with the cursor positioned at the start of the string. If the string could not be found, the message Not found will appear in the status area and the cursor will remain unmoved.

Whether `test` is treated as the same as `TEST` or `Test` etc. depends on which Casing button is selected. In the example above the search would stop if `TEXTWINDOWS` was found; if test<>Test was selected then the search would not find `TEXTWINDOWS` .

To find the next occurrence of the string click on Find Next from the Edit menu, or press Alt-N. The search starts at the position just past the cursor.

To search for the previous occurrence of the string click on Find Previous from the Search menu, or press Alt-P. The search starts at the position just before the cursor.

Having found an occurrence of the required text, it can be replaced with the replace string by clicking on Replace from the Edit menu, or by pressing Alt-R. Having replaced it, the editor will then search for the next occurrence.

If you wish to replace every occurrence of the find string with the replace string from the cursor position onwards, click on Replace All from the Edit menu. During the global replace the Esc key can be used to abort when the status area will show how many replacements were made. There is deliberately no keyboard equivalent for Replace All to prevent it being chosen accidentally.

To search and replace Tab characters press Ctrl-I when typing in the dialog box. Other control characters may be searched for in a similar manner except for the CR (Ctrl-M) and LF (Ctrl-J) characters. Alternatively, press Shift-Ins and this will display the character set from which you may pick the required character with the mouse.

# Disk Operations


The File menu contains many operations that involve using the disk system; you can save and load your source file, insert text into your source, delete a file from a disk and more.

## New

Select New to open an empty window, assuming that there is one available - you are allowed up to six windows at once in HiSoft BASIC.

Assuming that there are no more than five windows open, New will create a window which is empty and has no title.



## Loading Text

To load in a new text file, click on Load from the File menu, or press Alt-L. This will open a new window (and warn you if no more windows are available) or select an unused window and then a file selector will appear, allowing you to specify the disk and filename. Assuming you do not Cancel, the editor will attempt to load the file. If it will fit, the file is loaded into memory and the new window is re-drawn. If it will not fit an alert box will appear warning you, and you should use Preferences to make the edit buffer size larger, then try to load it again.

If the file can't be found a dialog box will appear, asking you if you wish to create that file. You may do so, or alternatively modify the filename and try again.

When loading HiSoft BASIC from a CLI, you may include up to six filenames. The corresponding files will then be loaded automatically. If a file cannot be found you will be asked if you wish to create it or may change the filename if you wish. If you use the desktop to install HiSoft BASIC as a *GEM takes parameters* (GTP) program then you may also enter up to six file names to be loaded.

## Revert

Revert will warn you that you are about to lose the text in the selected window and, assuming that you choose to continue, it will then re-load the last saved version of the file that you were editing in this window.

Revert will do nothing if you try to use it on a file that has not been saved previously.

## Save As...

To save the text you are editing, click on Save As... from the File menu, or press Alt-S. The File Selector will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing Return will then save the file onto the disk. If an error occurs a dialog will appear showing a TOS error number, as described in *Appendix C* of the *Technical Reference* manual..

If you click on Cancel the text will not be saved. Normally if a file exists with the same name it will be deleted and replaced with the new version, but if Make backups is selected from Preferences then any existing file will be renamed with the extension .BAK (deleting any existing .BAK file) before the new version is saved.

## Save

If you have already done a Save As (or a Load), the editor will remember the name of the file and display it in the title bar of the window. If you want to save it without having to bother with the file selector, you can click on Save on the File menu, or press Shift-Alt-S, and it will use the old name and save it as above. If you try to Save without having previously specified a filename you will be presented with the File Selector, as in Save As.

## Inserting Text

To read a file from disk and insert it at the current position in your text, click on Insert File from the File menu, or press Alt-I. The File Selector will appear and assuming that you do not cancel, the file will be read from the disk and inserted, memory permitting.

## Delete File

You may want to delete a file from disk (if for instance you have run out of disk space whilst trying to save); click on Delete File. The File Selector will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing Return will then delete the file from the disk. If an error occurs a dialog will appear showing a GEMDOS error number, the exact meaning of which can be found in *Appendix C* of the *Technical Reference* manual. If you click on Cancel the file will *not* be deleted.

## Close

This is the same as pressing Ctrl-W and will close the currently selected window. If the file that is being edited in this window has been changed since it was loaded or is a new file, you will be warned before the window is closed. You can choose to continue and lose your changes, cancel the action or save the changes.

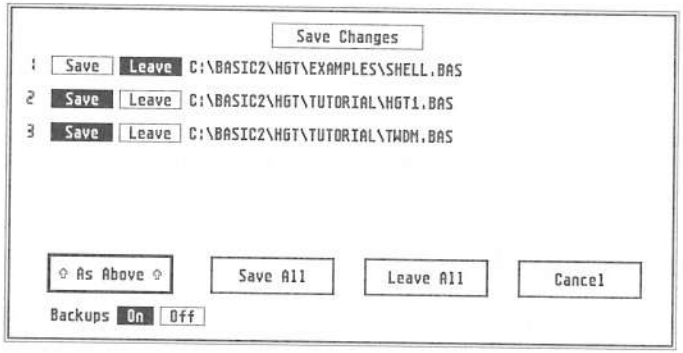


## Change Directory

This option allows you to move the current directory path; this can be useful when running programs which expect all of their files to be in the same place as the program itself. After clicking on Change Directory the File Selector will appear, allowing you to select a suitable disk and folder name. Clicking OK or pressing Return will then change the directory. If you click on Cancel the directory path will not be changed.

## Quitting HiSoft BASIC

To leave HiSoft BASIC, click on Quit from the File menu, or press Alt-Q. If changes have been made to the text which have not been saved to disk, an alert box will appear asking for confirmation.



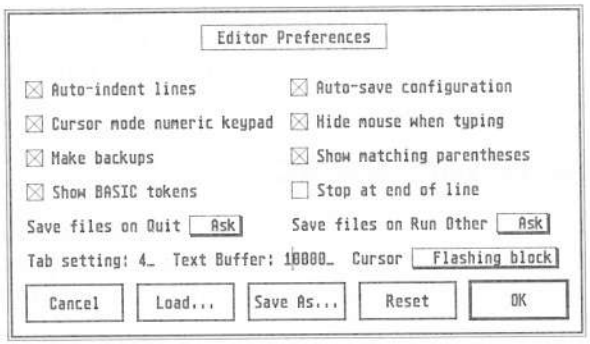

This example shows that two files have changed. Clicking on Save All, As Above or pressing Return will exit the editor saving the changes. Clicking on Cancel will return to the editor. Leave All will ignore all the changes you have made.

If you wish to save some files but not others click on the appropriate Leave buttons. For example if you clicked on the Leave button by SHELL.BAS in the above example and then pressed Return, only HGT1.BAS and TWDM.BAS would be saved.

You can also enable and disable backups from this dialog box. This is useful if you normally use backups, but decide that you don't require a backup of a one line change.

# Configuring the editor

Selecting Preferences... from the Options menu will produce a dialog box like this:




*The editor preferences box*

This box allows you to set up the editor as you would like to use it; you can then save your customisation to disk so that the editor will always behave the same way. Here are the different settings that you can change.

## Auto-indent lines

Selecting this option sets auto-indent mode. When active, an indent is added to the start of each new line created when you press Return. The contents of the indent of the new line is taken from the white space (i.e. tabs and/or spaces) at the start of the previous line. This allows you to lay out your program neatly, by simply pressing Return.

## Auto-save configuration

When this option is selected, the current preferences will automatically be saved when you exit the editor. So when you load the editor again, the preferences (including the compiler's options) will be just the same as when you last used it.



## Cursor Mode Numeric pad

The Cursor Mode Numeric Pad option allows the use of the numeric keypad in an IBM-PC-like way allowing single key presses for cursor functions, and defaults to Cursor pad mode. The keypad works as shown in diagram below:


When this option is not selected the keyboard reverts to returning the digits etc.

## Hide mouse when typing

Selecting Hide mouse when typing causes the mouse pointer to disappear when you start entering text with the keyboard. As soon as you move the mouse, or use a command that displays a dialog box, the mouse will re-appear. This option may be disabled if you prefer to always see the mouse on the screen.

## Make Backups

Selecting this option causes the editor to make a backup (with the extension .BAK) when saving files.

## Show matching parentheses

This facility lets you check that your parentheses match. With this option enables, when you press ) the cursor will quickly move to any matching ( character and then back to the current position, thus you can ensure that you have closed the correct number of brackets in a complex expression. If you find this cursor movement distracting then disable the option.

## Show BASIC tokens

Choosing this option will cause the editor to upper-case any HiSoft BASIC reserved words (see *Appendix A* of the *Technical Reference* manual) when you have finished editing a line.

This is very useful as a quick syntax check since you can see immediately which words you have mis-typed. It also makes your programs easier to read and to debug.

If this option is not set, the editor will do nothing with what you type, leaving reserved words as you entered them.

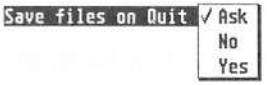
You can upper-case all the BASIC reserved words in a window, even if this option is not checked, by selecting Show Tokens from the Edit menu.

## Stop at End of Line

When this option is selected, if you press cursor left at the beginning of a line or cursor right at the end of line, the cursor does not move. Disabling this option, causes the cursor to move to the previous line if you press cursor left at the beginning, and to the next line if you press cursor right at the end.

The best way to find out which you prefer is to try using each setting.

## Save files on Quit




By default the editor will prompt you, if you are about to quit without having saved all the files, you have changed. The saving of these files can be made automatic by selecting Yes or disabled by selecting No (but don't blame us if you forget to save your files!).

## Save files on run other

This enables you to choose whether files are saved before using the Run Other and Run with Shell commands, in the same way as that for Save files on Quit.



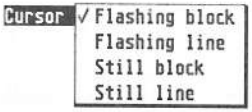
## Tab setting

By default, the tab setting is 4, but this may be changed to any value from 2 to 16.

## Text Buffer

By default the text buffer size is 10000 bytes, but this can be changed from 4000 to 999000 bytes. This determines the largest file size that can be loaded and edited. This amount of memory is allocated for each window in use. Care should be taken to leave sufficient room in memory for compilations - pressing the Help key displays free system memory, and for compilations this should always be at least 100k bytes. Changing the editor workspace size will cause any text you are currently editing to be lost, so a confirmation is required if it has not been saved.

## Cursor




By default the editor cursor is a flashing block, but this can be changed as required.

## Load ...

This button lets you load a settings file. The editor settings are normally stored in a file called HISOFTED.INF in the current directory, but the editor will 'look down' both the AES and GEMDOS paths. If you want to use more than one set of preferences, then you can explicitly load a settings file.

## Saving preferences

To save the settings file you can either choose Save as... from the Preferences box or choose Save preferences from the Options menu.

This latter command, on the Options menu, saves the current editor, compilation and Tools menu preferences under the name HISOFTED.INF. If you want to call your settings file a different name you should use Save as... in the Preferences... box, as described below.

When the editor is loaded, it looks for the HISOFTED.INF configuration file firstly in the current directory (which is the folder where you double-clicked on the data file), then using the system path. Saving the editor preferences this way will put the .INF file in the same place it was loaded from or, if it was not found, it will be placed in the current directory path.

In addition to saving the editor configuration the current program buffer size, from within the compilation options dialog box, is also saved.

Use Save as... from the Preferences box to save a settings file with a name other than HISOFTED.INF; an extension of .INF is still usual.

With this option you can save a number of different settings files under different names; however the editor always loads the settings file called HISOFTED.INF when it starts up so that, if you want to make a particular settings file the default, you will need to re-name it to HISOFTED.INF.

### Reset

Clicking on this box causes the settings to be reset to their default values; useful if you have made a complete mess of your options.

## Running other programs

There are three ways that you can execute other programs from within the editor; Run Other..., Run with Shell... and by a selection from the Tools menu. These different methods will now be described.



### Run Other...

This command, on the Program menu (also reached by Alt-0), lets you run other programs from within the editor, then return to it when they finish.

When you select Run Other... you will first be warned if you have not saved your source code (unless you have modified the setting of the Save files on Run Other option in Preferences). Then the GEM File Selector will appear, from which you should select the program you wish to run. If it is a .TOS or .TTP program you will be prompted for a command line, and then the screen will be initialised suitably.

This is the command to use for 'one-off' execution of a program within the editor. If you are likely to want to run the same program a number of times, then use the facilities of the Tools menu. If you would prefer to specify the program to run via a command line, rather than using the File Selector then use the Run with Shell command described below.

If you include the character sequence `%.` (i.e. per cent followed by full stop) in the command line (remember, you are prompted for a command line) these characters will be replaced by the full name of the file that you are currently editing. To pass the name without its extension, use `%?`.

If you need a true % to be passed type `%%`.

### Run with Shell...

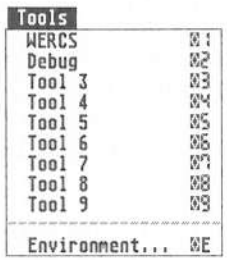
This command also lets you run other programs from within the editor, then return to it when they finish. The keyboard shortcut for this command is Shift-Alt-0.

It differs from Run Other in that you enter the file to run as a command line. If the editor finds that the `_shell_p` vector has been set up then this will be called to execute the command. This works well with the Craft, PKS and Gulam shells as the shell can be used to run batch files and expand file wildcards etc.

If the `_shell_p` vector has not been set up then the editor will look for the file to run using the PATH environment variable, which can be set using the Environment command from the Tools menu.

The same expansion of the current filename as used by Run Other can be used by this command. If you wish to use the same command more than once you will probably save time by using the Tools menu.

## Tools Menu



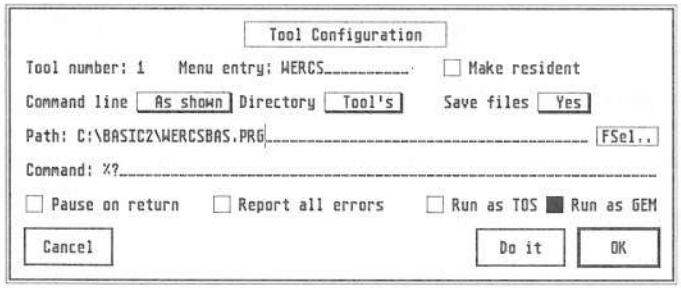

The Tools menu lets you run programs of your choice from within the editor using a single keystroke or click of the mouse.

The configuration can be saved in the preferences file, ensuring that the same facilities can be used again, the next time that you run the editor.

The preferences file that we supply is already set up to run the tools supplied with HiSoft BASIC.

Before you can use this facility you will need to configure each tool so that the editor can find the appropriate file. To configure a tool, hold down the Ctrl key and select the appropriate menu item or press Ctrl-Alt and the appropriate key *on the numeric keypad*.

This will produce a dialog box like this:




If you just want to use the default settings, you need only change the Path item so that the file can be found; either amend this item or click on FSel and use the file selector to select the appropriate file.



Once you have made the required changes you should press Return (or click on OK) to make your changes permanent; alternatively pressing Cancel will ignore any changes you have made. The other options in this box are:

### Menu entry

The name typed in this field gives the name of the tool as placed on the Tools menu. Hence in the above example the name WERCS appears on the menu.

### Command line


These options configure the way the command line is obtained for a program which is about to be run.

If None is selected then a program will be run as a plain GEM or TOS program with no command line. If Prompt has been selected you will be prompted for a command line in the same way as occurs when using Run Other.

Finally As shown allows the command line on the line below to be used. This command line is specified in the same way as that used by Run with Shell and may have the same meta-characters in it, as in the example above.

### Directory


This sets up which directory will be the current one when the tool is run. Current will leave the directory as that of the editor itself.

Tool's switches to the directory of the tool being run, whereas Top window switches to where the file in the current window is stored on disk.

### Save Files

This option changes which files will be saved before running the tool. If you select No then no files will be saved, selecting Yes (the default) will save *all* files (not just the current window), whilst Ask... will prompt you using the Save/Leave dialog described under Quitting HiSoft BASIC.

### Path

This option specifies which program is actually to be run. If you give a full pathname, or select one by clicking on the Fsel.. button then that specific file is run. If you just use a name then this will be treated as if you had used it as an argument to the Run with Shell command described above.

### Pause on return

This option controls whether the editor pauses after running the tool. Typically you will select this when running a TOS program but disable it when running a GEM program.

### Report all errors

This option allows you to specify which errors the editor will bring to your attention when returning. If this option is not selected then you will only be alerted to negative return codes from programs, i.e. those normally indicating GEMDOS errors. Selecting it will also force positive program error returns to be flagged.

### Run as TOS & Run as GEM

These buttons select how the program is run, either as a GEM program or as a TOS program; note that the same warnings about GEM/TOS mode made under Run with GEM apply here also.

### Make Resident

If this item is selected then when the editor next loads it will attempt to load this tool into memory and make it resident, i.e. merely execute the tool from memory rather than load it from disk each time. This is particularly useful with substantial programs like WERCS.

As well as the obvious disadvantage of permanently tying up your memory permanently, not all programs can be made resident. In general your own HiSoft BASIC programs can be made resident, but if you are writing mixed language programs. please see *Appendix F* of the *Technical Reference* manual. This also includes information on our other products that may be made resident.

We do not recommending running third party programs in this way. They may crash immediately, or the second time they are run or may simple not quite work correctly possibly destroying your valuable files in the process.



### Running Tools

Running a configured tool is simple, just select the appropriate menu item or press Alt and the appropriate key *on the numeric keypad* and the program will be run using the settings described above.

## Setting the Path

The editor maintains a number of directory paths to make the operation of the integrated environment natural and seamless.

Paths are routes to files. Normally you keep all files of a similar type or usage in one folder or you may have a number of related folders all within one outer folder. For example you probably have an HBASIC folder containing the HiSoft BASIC program, its tools and its libraries.

In order that a program that uses these files can find them without having to ask the user for help, both the ST/TT operating system and the HiSoft BASIC editor maintain a number of directory paths, some of which you can alter.

Here is a summary of the paths used by the integrated environment, how they are set and what uses them:

*Current directory* - this is a path that is set up (initially) by the program which ran the current program. For example, for the HiSoft BASIC editor this path will have been set up by the Desktop, assuming of course you ran BASIC from the Desktop. However, since the editor allows this to be changed (via the Change Directory command on the File menu), it is normally reset to whatever was last stored in the HISOFTED.INF file, to save you having to change it every time you run the editor.

Most of the disk-related functions within the editor will search this path first.

*GEMDOS path* - this path is that contained in the PATH environment variable. It is used by shells (e.g. Craft, PKS Shell, Gulam) to locate programs to run. It is specified as a list of , or ; separated folder names, each of which specify a folder which should be searched when trying to locate a file.

Within the editor it is used by Run with Shell, to locate the named program, and initially when attempting to find the HBASIC.LIB file. Other tools, like WERCS, may use it for locating subsidiary files, such as WERCS.RSC and WERCS.INF.

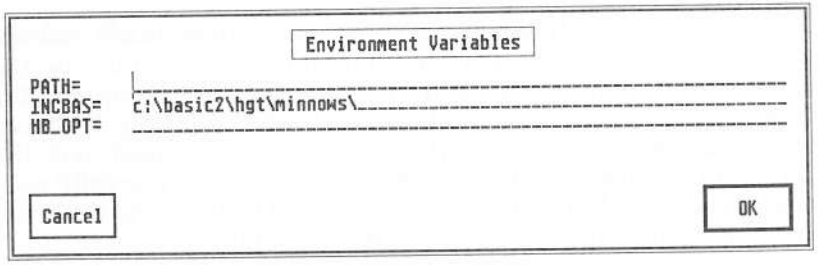
*AES path* - this is the path used by the AES when the user calls one of the AES routines which search for a file (`shel_find` and `rsrc_load`). Internally the format of this variable is identical to the GEMDOS path (in fact it *is* the GEMDOS PATH for the AES program!), although the AES provides no way of altering it and merely sets it to `A:\` for a floppy based machine or `C:\` for a hard disk machine.

*INCBAS path* - only used by the compiler for source includes and for .T files.

## Environment

The Environment option allows the environment variables used by the tools which are run (and other parts of the HiSoft BASIC system as described above) to be altered.

These variable names and their values are made available to any tools run from the editor so that the tools can find files and read the HiSoft BASIC options.




The settings displayed may then be altered to reflect any changes you may wish to make.

The PATH environment variable is discussed above, it is a comma- or semi-colon-separated list of paths which will be used by WERCS and command-line shells to find their associated files.

The INCBAS variable is a list of paths used by the compiler when it encounters the `$INCLUDE` option.

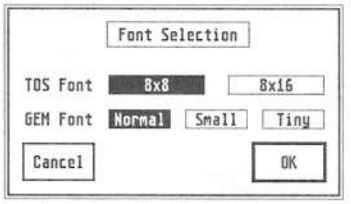


The HB_OPT environment variable lets you set the compiler's default options as if you were using the command line. This is for the use of any tool that may wish to read it. If you are compiling from the editor it is much easier to use the compiler's option box. Please note that this is *not* a path!

# Miscellaneous Commands

## Fonts...

The Fonts command is used to select different GEM or TOS fonts for use in the editor; it can be selected either by clicking on Fonts... from the Options menu, or by pressing Ctrl-G. It displays a dialog box like this:




The GEM Font is the font that will be used by the editor to display text. In ST high resolution and the TT resolutions, there are three fonts available as above. Changing to Small will double the number of line displayed on the screen. With the Tiny font the characters are only 6 pixels by 6 pixels wide but this does mean that even in ST high resolution, there are over 100 characters per line and 54 lines!

In ST medium resolution, there are only two fonts; Normal and Small. Small is 6 by 6 pixels and thus the characters are difficult to read but this does give an extra 7 lines of text and over 100 characters per line.

TOS font is used by non-GEM programs. In TT medium resolution, using 8x8 will give 60 lines instead of 30.

You should be aware that any change of font that you make here will also be effective outside the editor, after you leave it.

## ASCII Table...

To be found on the Edit menu, this displays a pop-up dialog box at the current mouse position, showing all the ASCII characters:

You may click on an individual character and it will be added to the text that you are editing at the current cursor position. You can bring up this display from the keyboard using Shift-Insert. This short cut can also be used in the editor's dialog boxes.

Note that the characters that would confuse the editor are 'greyed out' and may not be selected. Remember that characters other than those in the standard 7 bit ASCII set are not necessarily the same on other computers.

## About HiSoft BASIC

It you select About HiSoft BASIC... from the Desk menu, a dialog box will appear giving various details about HiSoft BASIC, including its version number. You will also be told the amount of free memory that is available to you and how much is used by the resident programs including the text in the open windows.


Pressing Return or clicking on OK will return you to the editor.



## Help Screen

The key equivalents for the commands not found in menus can be seen by pressing the Help key, or Alt-H. A dialog box will appear showing the cursor and function keys, as well as the free memory left for the system.

# Desk Accessories

If your system has any desk accessories, you will find them in the Desk menu. If they use their own window, as Control Panel does, you will find that you can control which window is at the front by clicking on the one you require.

For example, if you have selected the Control Panel it will appear in the middle of the screen, on top of the editor window. You can then move it around and, if you wish it to lie 'behind' the editor window, you can do it by clicking on an editor window, which brings the editor window to the front; you can then re-size it so you can see some part of the control panel's window behind it. When you want to bring the control panel back to the front just click on it and the editor window will go behind. The editor's cursor only flashes and the menus only work when an editor window is at the front.

# Automatic Launching

You may configure HiSoft BASIC to be loaded automatically whenever a source file is double-clicked from the Desktop, using the *Install Application* option.

To do this you first have to decide on the extension you are going to use for your files, which we recommend to be .BAS for BASIC files. Having done this, go to the Desktop, and click once on HBASIC.PRG to highlight it. Next click on Install Application from the `Options` menu and a dialog box will appear. You should set the Document Type to be BAS (or whatever you require), and leave the GEM radio button selected. Finally click on the OK button (if you press `Return` it will be taken as `Cancel`).

Having done this, you will return to the Desktop. To test the installation, double-click on a file with the chosen extension which must be on the same disk and in the same folder as HiSoft BASIC and the Desktop will load HiSoft BASIC, which will in turn load in the file of your choice ready for editing or compilation.

Note: To make the configuration permanent, you have to use the Save Desktop option.

# Compiling Programs

Having produced your BASIC program and saved it to disk using the editor you can then compile it to memory, compile it to disk or run it. Normally, when you choose to run a program from within the integrated environment, the program will be compiled first unless it has already been compiled successfully.

When compiling, you can use a large number of compiler options to affect how the compilation takes place - these options will be described below.

If you are compiling outside the integrated environment you will not be able to compile to memory and you will need to set the compiler options by either including them directly within your source code (via the `REM $option` directive) or by including them within the command line passed to the compiler. More details will be given below.

Here is an overview of the compilation process:

- Set up the compiler options which you can do within the program itself (via `REM $option`) or on the command line or from within the Compile box. See *Compiler Options* below.
- Ensure that the various file paths are set correctly - see *Setting the Path* above.
- Ensure that any include files and pre-tokenised (.T) files exist in the correct directory. See *Pre-tokenising* and *Compiler Meta-commands* below.
- Invoke the compiler directly from the integrated environment (using Alt-C, Compile or Alt-X, Run), from the desktop or from your favourite shell. If you are running from the editor, you can choose whether to compile to memory or not.



Compilation will then proceed and errors will be reported as and when they occur. You will have the choice of aborting the compilation and returning to the editor/shell/desktop or continuing. If the compilation finishes without errors you will then be able to run the program from memory or disk, depending on which you chose for the output.

We shall now explain pre-tokenising, include files, compiler errors, and compiler options in more detail.

# Pre-tokenising

Pre-tokenising of files is a method to speed up compilation of programs where a portion is not frequently updated, such as the GEM toolbox that we provide. There's a special command in the editor (Dump Tokens on the Program menu) which performs just the tokenising phase of the compiler and then saves to disk the current state of most of the compiler's internal tables.

These can then be loaded subsequently (via the Pre-tokenised file option) replacing the standard initialisation of those internal tables. This saves the bother of the most time consuming part of the compilation process.

When you choose to pre-tokenise a program in this way, the compiler will produce a file with the same filename as the source code but with a file extension of .T.

Then, if you specify a file name next to Pre-tokenised file within the Compile... box, the compiler will load the tokens found in this file (using the path specified by the `INCBAS` environment variable, then the current directory path) before compiling the rest of your program.

If you are using the stand-alone version of the compiler you should use the `$` and `@` compiler options to dump the tokens of a file and include a tokenised file at compile time respectively. See *Compiler Options* below for more detail.

There are a some restricions of which you need to be aware when using pre-tokenised files:

- The pre-tokenised file is parsed first - none of your main program will be considered until the tokens have been loaded.
- Pre-tokenised files may not contain DATA statements; the technical reason for this is that the code for DATA statements is actually generated during the tokenisation phase and the pre-tokenised files cannot contain code.
- You must not dump the tokens for a file, change HBASIC.LIB using BUILDLIB and then try to use the old pre-tokenised file. You must first re-tokenise any such files, otherwise your calls to the library may not work!
- You cannot use two pre-tokenised files at once; however you can use the pre-tokenise option in conjunction with Dump Tokens to effectively extend a tokenised file.
- Note that the default types for variables (DEFINT etc) and the switchable options are set up according to the pre-tokenised file.

# Compiler Meta-Commands

Meta-commands are special compiler commands that cause things to happen during the compilation - they do not produce instructions directly in the compiled code. Meta-commands are specified in the source code of your program by following a REM statement (or quote ' ) with a dollar sign.

## REM $INCLUDE filename

This command lets you include another BASIC source file within the current compilation, just as if it were there in the source. This can be used for using common modules between different programs, or for breaking up larger programs into modules.

## REM $DYNAMIC

Normally arrays that are DIMensioned using a constant value, for example `DIM A(100)` are treated as *static* arrays. Static arrays may not be re-dimensioned or erased which means the compiler can generate code that will access them very quickly.



If you have an array that you will subsequently want to re-dimension or erase then you'll need to use `REM $dynamic` which causes the compiler to make all DIMension statements declare dynamic arrays from this point in your program.

For example:

```
REM $dynamic
CONST arrsize=200
DIM arr(arrsize)
...
REDIM arrsize(300)
```

You can revert to the normal behaviour by using `REM $STATIC`.

Note that you must have used `REM $DYNAMIC` before you DIM the array for the first time; you cannot change the type of an array after it is declared.

Static arrays are stored in the program's global space rather than on the heap.

## REM $STATIC

This causes the compiler to revert to its standard treatment of DIM statements after you have used `REM $DYNAMIC`. See above.

## REM $OPTION option_list

This command lets you specify various compiler options. Each option is denoted by a letter then, optionally, a + sign to indicate on, or a - sign to indicate off, or a number in the case of some options. Individual options should be separated by commas. For example the line

```
REM $OPTION o+,a-
```

turns overflow checks on and array checks off. Options contained within programs like this override any chosen at the time of compilation, via the Compile... dialog box. The available options are discussed below under *Compiler Options*.

## Compilation Errors

When the compiler detects an error or something that may be an error (a warning) it generates a message; these errors are remembered, and can be recalled from the editor.

The message is prefixed by the error number and followed by the line and file in which the error was detected. In the case of warnings the compiler will continue automatically. After an error it will ask you if you wish to continue. If you type n or N (for no) you will be returned to the editor. If you hit any other key compilation will continue.

When you return to be editor you can use Alt-J to move to the next error with the error message displayed in the status line. If you have a large number of errors the editor may not be able to remember them all. Alt-J goes to the next error regardless of the position of the cursor; it will switch windows is required. To go to a previous error use Ctrl-J. The editor takes account of any insertions or deletions automatically so that unless one error (like a mistake in a definition) has caused multiple errors you should only need to compile once.

There's also the Shift-Alt-J command which finds the next error after the cursor in the current window. It is the appropriate one to use if you have got a number of include files and want to fix all the errors in one file before going on to the next one. You can also use it to find the first error in a file by typing Alt-T (to go to the top) and then Shift-Alt-J.

Occasionally the compiler will spot errors somewhat later than you might expect. This is usually because the text up to the point it has read is allowed in a certain context. If you have missed something out at the end of a line, then the error may be detected at the beginning of the next line. Note that, except in the case of missing sub-programs, line numbers or labels, the error in your program can not be *after* the point where the error was detected.

On occasions the compiler will generate more than one error message as a result of a single error in your program; do not be put off by this. If you get confused, just re-compile.

If you have a very badly formed source file, the compiler may slow down considerably. It is probably a good idea to type n to the continue prompt.



Incidentally, if you start a compilation of a large program you can break out and returned to the editor using the key combination Alt-Shift-Help when using the integrated compiler.

See the relevant Appendix in the *Technical Reference* manual for details of all the error messages and numbers produced.

## Compilation Options

The compiler options that affect the compilation process can be set via the Compile... box (when compiling from the editor) or via `REM $option` statements within the source code or from the command line if compiling with the stand-alone compiler (see *Stand-alone TTP Compiler* below).

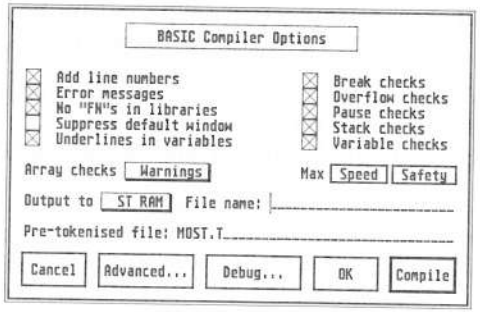
We shall discuss all these options below, initially with reference to the Compile... box; the option letter which is to be included after the `REM $option` line if setting options from within the source code, will be given at the end of the title description e.g. *Add Line Numbers (N)* means that the line:

```
REM $option N
```

is equivalent to setting the Add Line Numbers box in the Compile... box from within the editor. Options set in the source over-ride those set within the Compile... box.

### The Compile box

To set all compilation options and compile the program that is in the selected window choose Compile... from the Program menu or press Alt-C; a dialog box will appear:




*The Compile... dialog box*

The options towards the top of the box are connected with code generation whilst the remainder affect the more general behaviour of the compiler.

In general, most options, when used from the command line or after a `REM $option` statement should be followed by a plus (+) to turn the option on or a minus (-) to turn it off.

## Add Line Numbers (N)

This option adds line number information into your program. With the option on, after a run-time error the physical line number and the source file name will be printed if you haven't used `ON ERROR GOTO`. The physical line number will correspond directly to the lines in your source program. Program size is increased by 6 bytes for each non-blank line and there is a resulting degradation in program speed.

If you have used `ON ERROR GOTO`, `ERL` will contain the logical line number unless this would be zero (if you have no line numbers in your program, for example) when `ERL` will contain the physical line number. If you are not using include files this will correspond to the line in your source program but if you are using include files then it will correspond to the line number as if you had manually inserted your include files into your main program.

This option may be switched on and off throughout your program.

## Error Messages (E)

This option tells the compiler to include within the compiled program a list of textual error messages to be produced after a run-time error occurs. With the option off just a number will be printed in the event of a run-time error, which can be looked up in *Appendix C* of the *Technical Reference* Manual.

When the option is on the program size is increased by a small amount, but there is no speed penalty. This option is useful in desk accessories where code size is important and a runtime error spells disaster in any case.



## No "FN"s in libraries (#)

If this option is selected, there is no need to specify `FN` in front of calls to function in the operating system (or any user libraries) giving cleaner looking programs. The disadvantages are that the casual reader might mistake a function call for a variable reference. For example

```
print sversion%
```

will print the GEMDOS version number if you have used

```
library "gemdos"
```

whereas it will print the value of the variable `sversion%` if you haven't used this library. You will need to disable this option if you are compiling programs that use the libraries that were designed for use with previous versions of HiSoft BASIC.

## Suppress Default Window (Y)

If the Y+ option is specified then the default window will not be produced at the beginning of a GEM program. It is then the programmer's responsibility to open window 2 before trying to output anything to it (via the PRINT statement for example). If you are producing a .TOS program, then this option will suppress the clearing of the screen at the start and as such is ideal for writing utilities that are to be run from command line interpreters.

## Underlines in variables (U)

This is an option to maintain compatibility with Microsoft BASIC. With this option on, HiSoft BASIC allows underlines in variables and procedure names. With the option off, underlines can be used immediately after a reserved word or identifier, Microsoft style. See the *Concepts* chapter for more information.

## Array Checks (A)

With array checks on all array accesses are checked to have the correct number of dimensions and subscripts are checked to ensure they are within the range specified in the `DIM` statement. In addition any reference to an un-dimensioned array will dimension it.

For example the program segment

```
DIM a%(10)
a%(20)=10
```

with checks on produces an `subscript out of range` error.

With array checks off there are no checks at all, even on the existence of arrays. If you use an invalid subscript you are likely to destroy something else in memory, and if you try to use un-DIMmed arrays you will get unpredictable results, normally bus errors (2 bombs). There is an appreciable speed improvement (though an increase in program size) with checks off, particularly with one-dimensional integer arrays, but this should only be done when you are confident that your program works perfectly.

Destroying random areas of memory may not be immediately noticeable and may manifest itself some time later in unexpected ways. If you turn array checks off then `OPTION BASE 0` is forced throughout your program.

### Array Warnings ([)

Using this option will keep array checks on but will additionally warn you if the compiler automatically dimensions an array. You will therefore be told if you mistype the name of an array in a `DIM` statement.

Use this if you find that your program works with array checks on, but not with array checks off. If you are imbedding this option in your program or using it on the command line make sure that you use both A+ for array checks and [+ for the warnings.

## Break Checks (B)

This option allows a user to break out of a compiled program at certain times. With the option on Ctrl-C will abort a running program. Checks are made by the following actions: `PRINT` statements to the screen (regardless of the setting of pause checks), `INPUT` statements and the `INKEY$` function. If you want to allow the program user to break out at other times include a line such as

```
dummy$=INKEY$
```

at suitable places in your program.



This option may be switched on and off throughout your program, thus enabling you to disable break checks in pieces of code that you wish to have completed under any circumstances.

## Overflow Checks (O)

An overflow normally occurs when a numeric value exceeds its specified range. These ranges are shown in the following table:

| | | |
|---|---|---|
| % | integer | -32768 to 32767 |
| & | long | -2147483648 to 2147483647 |
| ! | single | -9.2E18 to +9.2E18 |
| # | double | -1.8D308 to 1.8D308 |

For example, the statement

```
i%=32000+10000
```

will produce an overflow as 42000 is too large to fit into an integer. With the `O` option on an extra 2 bytes are used for each maths operator and some I/O operations and there is a slight speed degradation.

The option should be turned off only after you are sure that your program is bug free, although you may use it selectively to disable checks in completed parts of your program. With checks off the results of calculations that result in overflow are not defined.

## Pause Checks (P)

This option allows a program's output via `PRINT` statements to be paused in a similar way to MS-DOS and CP/M. With the option on pressing Ctrl-S will pause output, and Ctrl-Q will resume it. This works for both TOS and GEM programs. There is no noticeable speed degradation or program size difference with this option on.

## Stack Checks (X)

With this option on checks are made before a `RETURN` statement is executed to ensure that it is sensible to do so; if not the error `RETURN without GOSUB` will occur. With this option off, no checks are made and a `RETURN` made out of context will cause unpredictable results.

## Variable Checks (V)

This option makes the compiler look for undeclared variables, a common source of bugs in BASIC programs. This only works for variables referenced within sub-programs or functions. For example, this program was mis-typed and with variable checks on the programmer will be alerted:

```
SUB initialise(mem%)
SHARED values%()
LOCAL i%

DIM values%(mem%)
FOR i%=1 TO mum%          ' deliberate mistake
values%(i%)=i%
NEXT i%

END SUB
```

Checks should be normally be on if you are compiling structured programs, but old fashioned BASIC programs should have the option de-selected.

## Output options

Essentially, HiSoft BASIC can compile to disk or to memory - compiling to memory is much faster and ideal for trying things out quickly, while compiling to disk means you can create stand-alone, double-clickable programs without any need for the compiler to be present. Simply select which option you want from the Output to pop-up menu.


If you are compiling to memory, you should set the size of the memory buffer that will be used by selecting Maximum... and entering a value in Kb; the default value is 30Kb.

This gives the size of your program's object code; the heap, which holds your program's variables, is *separate* and can be controlled using the k (keep) and l (leave) compiler options.



If you get a `program buffer full` error when you compile a program you should change the memory buffer to be larger. There is of course a penalty for this - the larger the program buffer size the smaller the amount of memory left for the compiler itself to use while compiling your program. If the compiler itself aborts with `Out of memory` it means there is not enough left for a complete compilation - you should reduce the buffer size, or if this still fails you will have to compile to disk.

When you compile to disk the program buffer size number is ignored, giving maximum room in memory for the compiler itself. The compiler will create an object file in the same path as your source file, shown in the title of the editor window unless you enter a specific file path next to File name. If you haven't saved your program yet and you do not specify a name next to File name, the object file will have the name NONAMEn (where n is the window number) and will be saved in the current directory.

The extension of the created file depends on what type of program you have compiled, and will be one of the following:

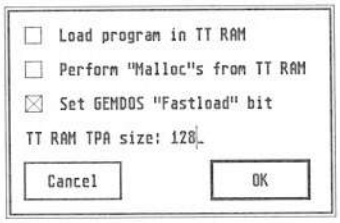
| | |
|---|---|
| .PRG | GEM program |
| .TOS | TOS program |
| .TTP | TOS Takes Parameters |

By default a program is .TOS, but use of the `COMMAND$` function changes it to a .TTP. Any use of graphics or GEM makes a program a .PRG file; you can also force a program to be a GEM program through use of the g compiler option.

The file produced on the disk can be run from the Desktop directly by double-clicking on it like any other program. It does not need any of the compiler or its library file to be present - it is completely self contained.

The stand-alone version of the compiler can only output to disk and so does not have an option for specifying the output destination in the above way.

## Load bits (^)




From TOS 1.04 onwards, four bytes in the header of every Atari executable program are used by the operating system to change the way that the program is treated when loaded. The Load bits... box lets you control these bits in the program header; the meaning of the various options are described below:

*Load program in TT RAM.* The program will be loaded into TT RAM if there is enough TT RAM available. You can use this for just about any HiSoft BASIC program. The exceptions are programs that change the physical screen address to point to an array or otherwise require an address used directly by the hardware to be in ST (system) RAM.

*Perform "Mallocs" from TT RAM.* `Malloc` calls will be satisfied using TT RAM if possible. Again you can enable this unless you are using direct hardware access to the memory that you are `mallocing`.

*Set GEMDOS "Fastload" bit.* If this is set, the whole TPA area past the end of the BSS is *not* zeroed. This results in decreased loading times on large memory machines for those programs. You can set this for any HiSoft BASIC program that doesn't assume that the memory that it `mallocs` using GEMDOS is zero.

*TT RAM TPA size.* The program's TPA size as a multiple of 128Kb. When the first option is set and this field is zero the program will be loaded into TT RAM if there is enough room for its code and if 128K of RAM is left for its variables and heap. If you would like your program to have at least 256K left, then make this field 1 and then, if there is insufficient TT RAM but enough ST RAM, your program will load there.



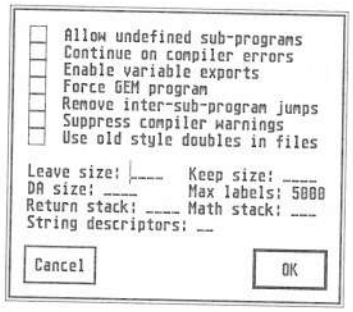
To set all the options on when using the stand-alone TTP version of the compiler you would use the following option:

```
rem $option %16777223
```

this corresponds to &h1000007. Using the interactive version is much easier - select the boxes and use 256 for the TT RAM TPA size!

# Advanced Options

Clicking on the Advanced... button in the compiler options box brings up a dialog box like this:




*The Advanced ...box*

We will now discuss these options, in turn:

## Allow undefined sub-programs (])

When using this option the compiler will not complain if you have declared a sub-program or function but not actually defined it. In general this is dangerous since if you actually call the missing sub-program your program will crash. However it is useful when writing general toolboxes that require user sub-programs to be called.

The GEM toolbox uses this option so that you don't need to write a sub-program to handle menu clicks if you are not using errors.

## Continue on compiler errors (C)

The `C` option causes the compilation to continue after an error without asking you. This is useful if you wish to leave a number of huge compilations taking place while you go and have a cup of coffee.

However if you are a beginner the extra error messages may be confusing (because the compiler itself can become a little confused) and you might prefer it to stop immediately if there is one error at the start of the file in any case.

### Enable variable exports (!)

This option is for use when linking with Devpac assembler language. When enabled it causes the names of all your global variables, sub-programs and functions to be exported. The use of these is described in *Appendix F* within the *Technical Reference* manual.

### Force GEM program (G)

This option forces a compiled program to be a GEM program, with all input and output occurring in a window; this will automatically happen if any graphics commands are used. You may wish to force GEM mode if your program could run as a TOS program but you would prefer to have its output in a GEM window.

### Remove inter-sub-progam jumps (%)

Normally HiSoft BASIC will add a jump instruction at the beginning of every sub-program and function so that if execution 'falls through' to the sub-program execution will continue at the next instruction after the sub-program:

```
GOTO Finished_John
SUB John
...
END SUB
Finished_John:
```

with the references to `Finished_John` added by the compiler.

Enabling this option will cause these jumps to be omitted, thus resulting in a smaller finished program especially if you have many small sub-programs. However you need to ensure that a sub-program cannot be executed, as a crash will probably result.

If your programs have sub-programs at the top and the main program at the bottom, you'll need to add a `GOTO` at the start of your main program in order to use this option.



### Suppress Compiler Warnings (W)

If the `W+` option is specified then the warning messages normally produced by the compiler will be suppressed.

### Use old style doubles in files (*)

HiSoft BASIC 1, Power BASIC and FirST BASIC for the ST normally store their doubles in binary files (using `MKD$` and `CVD`) in a different format to that used by HiSoft BASIC 2, HiSoft BASIC for the Amiga and most languages that use IEEE doubles for 68000 computers.

If you enable this option then HiSoft BASIC 2 will use the older style double ordering, meaning that you can use data files created with the older HiSoft BASICs for the ST.

In general, we suggest that you convert your data files to the new format since this not only makes your data compatible with other machines but also it is likely that this option will be removed in the future.

### Maximum Labels (H)

This sets the size of the code generator's label table. Using a larger number (the maximum is 5000) will generally give a faster compilation, although for small programs it doesn't have much effect. Each entry in the table requires 6 bytes and you'll receive a `Label table full` error message if you use too small a number.

### Return Stack Size (R)

The return stack is that used for return addresses after `GOSUB`s, function calls and sub-program calls. It is also used for local numeric variables and by the run-time system. If it should ever run out it will hit the maths stack, which will result in various forms of crashes.

The minimum is 200 bytes, and the default is 4096. If you have a heavily recursive program you may need to increase the return stack size.

### Maths Stack Size (M)

The maths stack is used for storing temporary numeric values during calculations and function calls. Should it ever run out the global area workspace will be corrupted. This option lets you set the maths stack size in bytes.

The minimum value is 32 bytes and the default is 256 bytes. You should only need to change this if you have incredibly complicated numeric expressions.

### Temporary String Descriptors (T)

In the highly unlikely event of the error `String expression too complex` appearing the number of temporary string descriptors may be increased from the default value of 15, for example:

```
rem $option T30
```

## Startup Options

There are three different versions of the startup code, determined by the use of the following meta-commands:

### Leave Size ( L)

This allows the programmer to determine the amount of memory returned to the system at startup, the remainder being used by the running program. This option is mainly useful for GEM programs as the system memory area is used for the following things:

- Resource files loaded with `rsrc_load`
- Window titles and status lines
- Menus created with the `MENU` function
- Windows opened
- Desk Accessory workspace
- Other programs loaded with GEMDOS `pexec` call
- Fonts and printer drivers loaded using GDOS

The default is 4k bytes, but can be made bigger or smaller as required. For example if you are loading a large resource file you will need to make it larger. The compiled program will itself use the rest of the available memory. For example

```
rem $option L40
```

would leave 40k bytes free.



### Keep size (K)

This option specifies that a program will use a particular amount of memory, the rest being returned to the system. If you try to keep more memory than there is in the machine an error message will appear during program initialisation and execution aborted.

This option would be useful for a menu program which in turn loads other programs and has a low memory requirement itself, so you might use:

```
rem $option K10
```

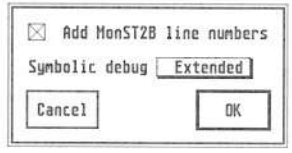
which would keep 10k bytes, returning the rest to the system.

### Desk Accessory Size (J)

This option works the same way as the Keep size (K) option but uses the desk accessory start up code. For information on writing desk accessories please see *Appendix G* of the *Technical Reference* manual.

## Debug Options

Clicking on Debug ... from the compiler options box will bring up another dialog, like this:




*The Debug ...box*

These options are used in conjunction with our medium level debugger, MonSTB or if you wish to use a low level debugger

### Add Debugger line numbers (&)

This option causes extra information to be output to your program so that you can use the # operator in MonSTB that gives the program counter corresponding to a given line number. This means that you can set breakpoints etc. using BASIC line numbers rather than using machine code addresses. The line number information averages just over 2 bytes for each line that contains executable code. Whilst this increases its size on disk and when run using the debugger it does not increase the memory used by the program when run normally.

This line number information is automatically included if you use the REM $PRON meta-command.

### Use Extended Debug (D)

This option causes HiSoft BASIC to output HiSoft Extended Debug giving 22 characters of significance for labels rather than the 8 characters used by the standard DRI format. When using the stand-alone compiler you need to use both this option and option S, below to actually output the symbols.

### Add Debug information (S)

Setting this option causes symbolic labels to be added to your program for your sub-programs, functions and the entry points to the runtime system for use with debuggers such as MonSTB. If using the interactive compiler, you will probably want to use the *Extended Debug* option (see above) instead. If you are using the stand-alone compiler, use both these options.

## Other options

### Load pre-tokenised file (@)

This option enables you to load a pre-tokenised file, like the GEM Toolbox, before compilation starts. This means that if you are using some debugged routines you can compile much more quickly. Pre-tokenised files are compiled using the Dump Tokens command from the file menu or via the $ option. This is described in detail below.



### Dump Tokens ($)

This command causes the compiler to only perform its tokenising phase and then to output a file containing the tokens and internal tables produced. You can then use the *Load pre-tokenised file* (@) option to speed up subsequent compilations of files that would otherwise need to *include* this code. See the section on pre-tokenised files for more detail.

The following options can only be used on the command line or embedded in the program. using REM $OPTION.

### Remove reserved words (~)

This option lets you disable specific reserved words so that you can then use them as variables; the corresponding command cannot be accessed subsequently. To remove more than one word at a time separate the reserved words by a comma but without any spaces. For example, the following piece of code

```
FOR loop=1 TO 10
PRINT loop,loop*loop
NEXT loop
```

will work on old-fashioned BASICs that don't support DO...LOOPs but won't with newer BASICs because loop is a reserved word. To get this to work with HiSoft BASIC you could change all the loops to another name using the Search and Replace commands of the editor, or you could add

```
REM $OPTION ~LOOP
```

to the start of the of the program.

To help people porting programs from other systems we supply a number of include files that remove all the HiSoft BASIC reserved words that can be used as variables in that dialect. Whilst this can help you get the program running, it does mean that you will not be able to use the extra features of HiSoft BASIC to enhance the program, so we recommend that you only use this if you are in a hurry!

### Finish without waiting for a keypress (Z)

This option causes the stand-alone version of the compiler not to output a message and wait for a key at the end of compilation. This is for the benefit of command line interpreter and make file users. If you are using a CLI you'll probably want to use this in the HB_OPT environment variable, so that it will apply to every compilation.

If you are using the command line version from the desktop, you won't want to use this option.

### Suppress compiler titles (.)

This option (a period) suppresses all compiler text output except error messages and is designed for CLI users who want the minimum output from multiple compilations.

## Stand Alone .TTP Compiler

In addition to the integrated, GEM-based version of the compiler we also supply HBASIC.TTP. This is a stand-alone, TOS version of the compiler intended for users who have committed editor preferences, or who prefer a CLI-type of programming environment.

To use this from the Desktop simply double-click on it and enter a suitable command line, of the form:

```
source_filename [options]
```

The source_filename should be the name of the BASIC source file requiring compilation, and .BAS is assumed as an extension, but may be set explicitly to be something else. The options should be specified starting with '-' and are the same as those described previously.

One important difference between this and the integrated compiler is that all options default to *off* in the .TTP version.

The file HBASIC.LIB should be on the same disk and in the same folder as the .TTP file.



For example, the command line:

```
DEMO -O+,B+,FTEST
```

will compile the file DEMO.BAS with overflow and break checks on and create the output file with the name TEST, the extension being decided by the compiler.

Users of batch files may wish to use the Z option, which obviates the need to press a key after successful compilation and returns immediately (Z stands for zzzz!).

Note that as with the integrated version the compiler will use the INCBAS environment variable to search for include and pre-tokenised files and the HB_OPT environment variable gives the default options, so CLI users may wish to set HB_OPT to Z to suppress the wait for a key message.

## TT version of the compiler

The TT version of the compiler works in a similar way to the standard version except that it uses and generates 68030 instructions for faster and more compact code and generates inline code for the 68882 maths co-processor. The code produced requires at least a 68020 and 68881 combination.

The code for the 68882 uses the full extended precision on chip and stores intermediate values on the maths stack as doubles regardless of whether they are stored in memory as singles or doubles. In general the TT version is more accurate than the software routines. Rather than storing singles as Motorola Fast Floating Point (FFP), IEEE singles are used as supported by the chip directly. These have different accuracy and precision to FFP numbers.

As such, you should not expect the same answers for numeric calculations on both compilers; if you are developing for both the ST and the TT you need to check the accuracy of your calculations on both machines.

We could have made the TT version produce results that are much closer to those produced by the ST version but the code would have executed much slowly since inline code could not have been produced. One floating point intensive benchmark runs 100 times faster with the TT version than it does on the ST version on the same machine.

For commercial developers who want their products to run on both machines, rather than providing an auto-sensing version of the system that would use software or hardware floating point according to the system, we suggest that you provide two versions as we have done with HiSoft BASIC; this really lets the TT version take advantage of the new hardware.

# Running Programs

If you click on Run from the Program menu or press Alt-X you will run a program previously compiled into memory. If you have made any changes to the source code of the current program since the last compilation, an automatic compilation will occur, before the program is run. If there are any compilation errors then it is not possible to run the program. When a compilation occurs as a result of a Run command it uses the current options as set by the last Compile... command.

Programs compiled to memory and executed using this command work in a very similar way to when they are compiled to disk and double-clicked, although there is one important difference - they have much less memory available when they run, as an editor and compiler (and maybe other tools) are sharing the memory space with them.

A program compiled to memory can be broken out of at most times by pressing the key combination Shift-Alt-Help.

## Run Directory

Selecting Directory on the Compile menu lets you choose the directory path that will be passed to any programs that you run in memory. If you select Current then the editor's current directory will be used; when Top Window is select the directory corresponding to the top most window will be used.



This can be useful if your program needs to load subsidiary files as it can find them immediately even if you have loaded the editor from another folder. Remember that you can change the editor's current directory using the Change Directory command from the File menu.

## Problems

When issuing a Run command from the editor the machine may seem to 'hang up' and not run the program. This occurs if the mouse is in the menu bar area of the screen and can be corrected by moving the mouse. Similarly when a program has finished running, the machine may not return to the editor. Again, moving the mouse will cure the problem. This is due to a feature of GEM beyond our control.

# WERCS
# The Resource Editor

WERCS is an acronym for WIMP Environment Resource Construction Set and is pronounced *Works*. It allows you to create and edit resource files for use with GEM programs and also provides you with the information you need to write these programs in HiSoft BASIC.

## What is a Resource File?

A resource file is a special file (normally with the extension .RSC) that contains *resources*. A resource is actually a *tree* structure in memory which is used by the GEM AES to produce such things as:

- *Menus*
- *Dialog boxes*
- *Icons*
- *Alert boxes*
- *Strings*

A resource file contains such things to deliberately keep them separate from your program code. In addition, the X-Y co-ordinates of every item in a tree is stored in such a way as to produce the same visual layout, regardless of the screen resolution. This means one resource file can be used for all screen modes and by many different programs.

Using resource files is good practice because it encourages modularity and aids portability thus saving you time and energy in the long run.

A certain understanding of the way a resource file works is required in order to create and use such a file.

Each resource file contains one or more *trees*. A tree may be one of five different types: *Form*, *Menu*, *Free String*, *Alert* or *Free Image*. Forms and Menus are the most common; each of these, in turn, consists of individual *objects*, where each object has a distinct type, use, purpose and appearance.



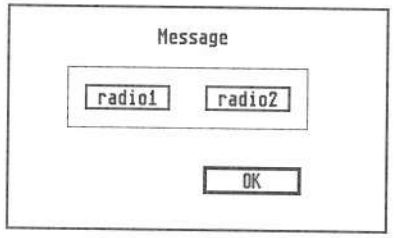
Whilst you are learning to use WERCS we recommend that you start off by using just Forms.

## What is a Tree ?

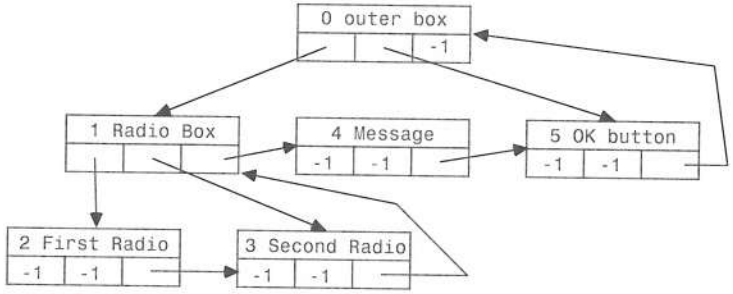
Forms (or Dialog Boxes) and Menus are GEM AES *object trees* and to understand resource files you need to understand the structure of object trees.

Many of the WERCS commands work on parts of object trees and we shall use tree terminology to describe them.

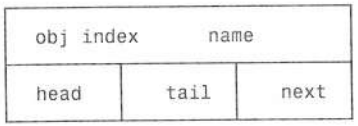
When it is loaded into memory an object tree is like an array of records, each record describing an object. The first object (with index 0) is called the *root* object. It is normally the outer box of a Dialog Box. Each object in the tree has eleven fields. Three of these fields, the *head*, *tail* and *next* fields, hold integer values that dictate to the AES the structure of the tree. Fortunately you do not normally need to access these directly, WERCS does it for you. As an example, say we have a Dialog Box like this:




The tree structure this represents can be shown as:



where the components of each box are:




Object number 0 is called the *root* of the tree. Its *children* are Message, Radio Box and OK button. Radio box's *parent* is outer box; its children are First Radio and Second Radio; its *siblings* are Message and OK button. First radio and Second radio are *childless* and they are *grand children* of the root object, outer box.

Most of the terminology used to describe object trees is similar to that used in human family trees; of course objects only have one parent and most people don't think of themselves as ultimately descended from a root!

Normally what is important with object trees is the tree structure, not the order that the items are in memory. The detail of how trees are stored in memory is described in *Technical Reference - Chapter 4*.

## What is an Object?

There are thirteen types of object that you can have in object trees; most of them are some form of text or boxes or a combination of both. Different types have different memory requirements; in general the more flexibility the more bytes are used.

As well as the fields described above, associated with each object is its *position*, *size* and also some *flags* and *states*.

The position of an object is always given relative to its parent; normally you set the position and size of the object using the mouse - WERCS takes care of the calculations for you.

The flags and states are used for two purposes; first to change the appearance of an item; for example whether a box has an outline (Outlined), and also to give information to the AES; for example that clicking on a Button will cause control to be transferred back to your program (Exit). To start off with you need not be too concerned about flags and states as WERCS gives you sensible defaults.



The thirteen types of objects are as follows:

- *Box* — A straightforward box - can have a fill pattern and a border.
- *IBox* — An 'invisible' Box; only truly invisible if it has no border.
- *String* — A straightforward string of characters.
- *Button* — Like a String but with a box round it; normally used for Dialog Box buttons.
- *Text* — Like a String but with more formatting possibilities: colour, size and justification.
- *BoxText* — Like Text but with a surrounding box as well.
- *BoxChar* — A single character in a box. The most memory efficient way to have a single character in a filled or coloured box.
- *Title* — A special form of String only used in Menus.
- *FText* — This is like Text but can be used for editable text so that you can type in characters, numbers etc. The programming interface isn't easy but we show you how to do it in the example program.
- *FBoxText* — Like FText but with a box around it.
- *Image* — A simple bit-mapped graphic image.
- *Icon* — Like an Image, but with a mask so that it changes sensibly when selected and also has a character and string associated with it. Originally invented for the desktop's disk icons.
- *ProgDef* — A programmer-defined object with its own drawing routine. We recommend that you don't try these out until you've exhausted the possibilities of the pre-defined objects.

## Header Files

In order for a program to use a resource file, the programmer must be given a method of referring to each tree and object; WERCS helps you by creating a *header file*, as well as the actual resource file. As you create a resource file you can give names to both trees and objects, so that you can refer to these names within your program. The header file contains constants which translate these names into integer values. The header file is made available to your program using `REM $include`.

If the compiled version of the header file is out of step with the resource file, strange things will happen; this varies from slight mis-behaviour to total system crashes.

# Quick Tour

## Running WERCS

To run WERCS, simply double-click on the WERCSBAS.PRG icon or run it from the Tools menu from within the HiSoft BASIC editor. WERCS also needs its .RSC file in order to run.

There now follows a whistle-stop tour of WERCS introducing the editing facilities available. A more detailed reference section is to be found in the next section.

### Low Resolution

WERCS runs in all screen modes, for maximum flexibility. When running in low-resolution, the title of each menu is reduced to the first two characters only. However, the full menu title is shown at the top of the menu box, once it has been pulled down.

## Creating a New Resource File

Having loaded and executed, WERCS will display a *tree window*, labelled Untitled. A tree window displays all the trees within the file and initially this is blank since you are starting with an empty file.

### Creating a New Tree

To create a new tree you simply select a suitable type of tree from the Tree menu - this stage is known as *tree-level editing*. An icon representing this type of tree will appear in the tree window, together with a dialog box. The main point of interest for the time being is the name that you wish to call the tree - when you are happy with its name press Return and you will then be at the *object-editing* level.



### Creating Objects

WERCS will now display a window showing the new tree, allowing you to add and edit new objects. To add an object, select the object type from the Object menu. The mouse will change into a representation of that object, then you should click where you require the object to be placed. To name this object, double-click on it, to move it simply drag it, or to re-size it click on its lower right-hand corner.

When you are satisfied with the objects in this tree, clicking on the *Close* box will return you to the main tree window. If you don't like anything you have done, the last session may be aborted by selecting Abandon Edit from the Edit menu.

When the tree window is visible, an existing tree may be edited by double-clicking on its icon. Certain attributes, such as the name, may be changed by single-clicking to produce a dialog box.

When you are happy with your resource file you can save it using Save As. This will create the .RSC file containing the actual resources, a .HRD file containing your names for each item, and a header file for use with HiSoft BASIC programs.

# Using WERCS

## General

Most of the editing actions inside WERCS are obtained from menus or via the corresponding keyboard shortcut. Keyboard shortcuts are shown in each menu with a ⌘ symbol denoting the Alt key, and ^ denoting the Ctrl key. Menus that are inapplicable at a particular time are disabled. Owing to a bug in the original version (1.0) of the operating system, the titles are not disabled when using these ROMs, although naturally these commands will have no effect. There is a summary of the keyboard short-cuts at the end of this chapter.

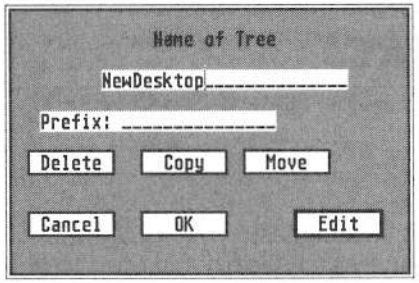
## Introduction to Creating and Editing Trees

There are two main levels when running WERCS. The *Tree Level* is used for manipulating the trees which are in your file and the *Object Level* for the items within those trees.

When you open a file (or use New) a window containing the trees in the file is shown, known as the *tree window*. You can add a new tree to the file by clicking on one of the items on the Tree menu: Form, Alert, Free String, Menu and Free Image. Whilst you are learning to use WERCS and if you are not familiar with GEM it is best to just use Forms. Forms account for the vast majority of trees in any case.

Note that a Form is also known as a Dialog Box; we use the terms inter-changeably. Generally we use Form in the context of editing and in the programming section we refer to Dialog Boxes.

After clicking on Form from the Tree menu you will be presented with a dialog box like this:




You can now enter the name of the tree. The defaults for these are MENU1, MENU2, FORM1, FORM2 etc.

Pressing the Return key or clicking on the Edit button will then let you edit the objects within the tree. The other buttons and fields are described in detail later.

To edit an existing tree, such as a Menu or a Form, double-click on the appropriate tree icon. You will then be taken straight to the Object Level.



## Changing Objects

Once you have clicked on Edit from the Name of Tree box you are ready to add objects to the tree. To add a new item to a tree, click on the required item type from the Object menu. The mouse will change to an outline representation of the item that you are adding. Release the mouse button and move the mouse to where you would like the new item, then press the mouse button. If you decide you do not want to add this item after all, click on the Cancel item from the Edit menu.

When you have finished editing a form click on the *Close* box; this will return you to Tree Level mode.

### Selecting objects

To change the attributes of any object, single-click on it. It will then be *selected* (highlighted) and you can use most of the menus to change its attributes, the border for example. The exceptions to this are the text items, Images and Icons; to edit these, double-click on an object.

When the object is selected the GEM *selected* bit is used to show this. This means that if you click on a box it will appear black. In particular if you click on the outer box it will all go black. If you didn't mean to click there you can either:

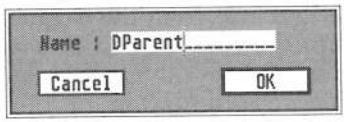
- click on the menu item Cancel from the Edit menu,
- click outside the box, or
- click on the item that you meant to select.

If you wish to edit the *parent* of the current object, single-click with the Alt key held down - the parent of the object will then be selected. If you already have an item selected and Alt-click again, its parent will be selected. This may be repeated any number of times until the whole tree is selected. To bring up the Text Box of an object's parent double-click whilst holding down Alt.

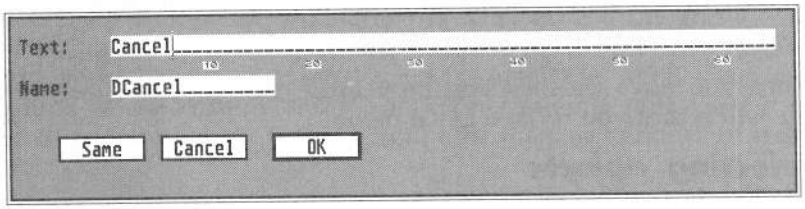
### Item Names and Text

To change the Text or Name (remember: Text is the displayed message; Name is what your program will know the item as) of an object, double-click on that item - this will present a Text dialog box. This varies depending on the item.

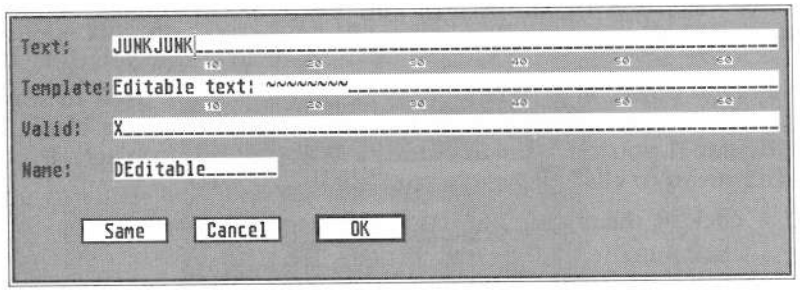
For example a Box only has a Name and presents a dialog box like this:




Buttons have one Text field and so have this type of box:




Whereas FText and FBoxText items have the appropriate TEDINFO fields as well:




To make the Name the same as the Text, click on the Same button - this is like clicking on OK except that the object will be given a name based on the text of that object and the Prefix for this tree, if any. This can save a great amount of tedious object naming. For editable text the name is taken from the Template thus giving the same name as your prompt.

Since underline characters are used by WERCS in a special way then, when editing text fields, you should enter underline characters (_) as tildes (~) and vice versa. If we didn't do this it would be impossible to see how many underlines you have in your Template strings.



To enter control characters into strings enter \\ (two back slashes) followed by the ASCII symbol corresponding to the control character. For example the Alt key symbol is entered as \\7. Similarly the copyright symbol (©) is \\189. Don't try and enter a null character \\0 as the AES treats this as the terminator of the string.

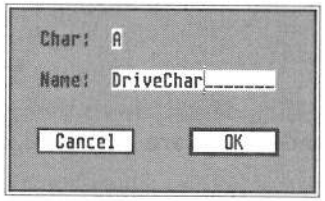
In the unlikely event that you need to enter two consecutive back-slashes type three instead; for three back-slashes type four and so on.

When using formatted text (FText or FBoxText) you should ensure that the Text field has the same number of characters as the Template field has ~s (stored in the file as underlines). If you are using different Valid characters then you should have the same number of characters in the Valid field as there are ~s in the Template field.

If you are using the same character throughout the Valid string you can enter just one as in the example above. We have not seen this facility officially documented but it works with all known versions of the operating system at the time of writing.

The other attributes of TEDINFOs (such as Large/Small characters) are set using the Text menu.

BoxChars (single characters surrounded by boxes) have their own dialog box, thus:




## Moving and Sizing Objects

To change the size of an item, place the mouse near its bottom right-hand corner and drag to the required size. By *near its bottom corner* we normally mean within *one character cell* of the bottom corner but inside the object. If the object is less than a character high then you should click within the bottom half-character of the box. Similarly if it is less than one character wide you need to click within a half-character of the right.

Note that the border of an object is often outside the object itself as is any outline or shadow of a box. This area is not considered part of the object by the `objc_find` call. This means that you will not be able to select or drag an object by clicking in its border, outline or shadow. Also, any program you write to handle such objects must bear this in mind.

To move an object within a tree, drag from somewhere other than the bottom right-hand corner. If the object has any children, they will move with the object.

To move or size any parent or siblings of an object, first select the required objects using Alt-clicking as described under *Selecting Objects* above and then drag as if you were moving a single item.

If you want a quick copy of an object or objects, select and then Shift-drag. Generally it is best to Shift-click with the mouse near the top left corner of the object as this is where the object will appear. This will let you move a new copy of the object leaving the old one where it was, so you can drag the new one to its new position. This is particularly useful if you have a set of similar objects with the same flags and attributes set (say disabled, right justified small Text). Set up the first one and then Shift-drag to create the rest.

If you using Shift-clicking to move an object which has children, you will get copies of the children too. When you let go of the mouse you will be asked whether you wish to delete the children too. Indicate that you do wish to delete the children; otherwise you will get an extra set of children, starting where you originally clicked.

If you move an object outside its parent then you will be asked for confirmation of this (unless you are in Expert Mode).

If you move an object so that it would completely cover another object or objects then you will again be asked if you wish to adopt these objects as children of the object that you have moved. If the new position of the object will partially cover another then you will be given an error message.

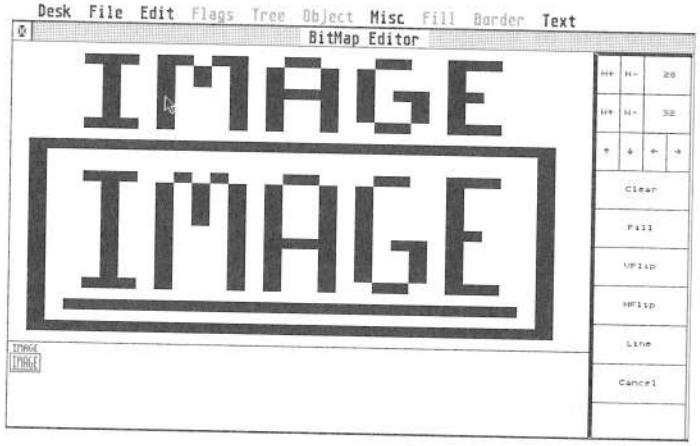
After you have moved or sized an object it may be snapped to the nearest character or half-character boundary, if you have used the Auto Snap or Half Character Snap commands.



You may also change the position and size of an object using the Extras command from the Flags menu.

## Editing Images

Double-clicking on an image will bring up the Icon Editor which will give a screen display similar that below:




The largest and main part of the display is used for editing the Image a pixel at a time. Beneath this is an Actual Size representation of the Image, as it will appear in your form and to the right are various buttons that you may click on.

To change an individual pixel, just click in the appropriate place on the screen; if it was black it will become white and vice-versa. To make a number of pixels the same colour click and drag; note that the actual size display will only the updated when you release the mouse button.

At the top of the button area are the buttons for changing the *height* of the Image together with the current height (28 in the example above). To increase the height click on H+ and to decrease it click on H-. Both of these work one pixel at a time and will repeat if you hold the mouse button down. The size of the main display changes to ensure that it is as large as possible whilst still displaying the Image at actual size beneath it.

If you decrease the height by too much, increase the height again and the newly displayed area will be the same as it was before you made the Image display smaller. The maximum height of Image that you can edit is 128 pixels. If you attempt to edit a larger image, it will be truncated to 128 pixels high.

To change the *width* of the Image click on the W+ and W- buttons; the current width is displayed to the right of these buttons. GEM restricts the size of Images to multiples of 16 pixels (so that it can draw them on the screen quickly) so these buttons change the width 16 pixels at a time. The width of the main display and the button will change to give as large a main area as possible. These buttons do not repeat if you hold them down. As with height changes, do not worry if you make the width too small by mistake, just click on W+ and the area that you have just deleted will re-appear.

The maximum width of an Image that can be edited is 128 pixels. Again, editing an image that is more than 128 pixels wide will cause it to be truncated.

The arrow buttons scroll the main display in the appropriate direction. Scrolling upwards and to the left loses the pixels that are removed from the Image. The pixels that are lost when scrolling to the right or downwards can be retrieved by scrolling to the left and upwards, assuming that the maximum size of 128x128 is not reached.

The Clear button will clear the entire Image to white; unless you are in Expert Mode you will be prompted to check that this is what you want.

The Fill button will fill the entire Image to black; as with Clear you will normally be prompted for confirmation.

VFlip and HFlip reflect the Image in a vertical/horizontal line through the middle of the Image. The best way to understand this is to try it. Clicking on VFlip (or HFlip) twice is like doing nothing at all.

Line is used to draw a line of black pixels. The mouse cursor will change to a +; click where you would like the line to start and then on where you would like the line to finish.



Cancel is used to cancel all the changes that you have made since entering the Image Editor; if you are not in Expert Mode you will be prompted to ensure that this is what you require.

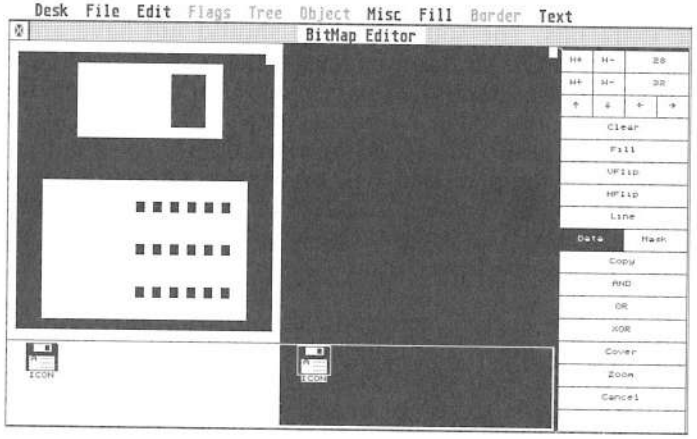
To name an Image, double-click in the Actual Size area; the usual name box will then be displayed.

The Text menu can be used to set the foreground colour of the Image.

The normal way to exit from the Image Editor is via the Close box although you can also use the commands on the File and Misc menus.

## Editing Icons

The display when editing an Icon is like this:




Editing an Icon is like editing an Image except that the main display consists of the Data of the Icon on the left and the Mask of the Icon on the right. There are also some extra buttons in the Icon Area, and the Icon's string and character fields can also be accessed.

There are two Actual Size displays; the one on the left normally shows the icon not selected; that on the right shows it selected. If however you set the Selected bit using the Flags menu from the Object Level window then these will be the other way round. The Icon on the left is always as it will appear in the file.

The extra buttons for Icons are used as follows:

Data and Mask are a pair of radio buttons; if Data is selected then the Data bit map is used as the source for the commands below and also as the current bitmap for Clear, Fill, VFlip and HFlip; WERCS will remind you which bitmap you are destroying unless you are in Expert Mode. The rest of the commands are described assuming that Data has been selected; to perform the action the other way click on Mask to select it first.

Copy, AND, OR, and XOR perform the appropriate logical operation; so that Copy will make the Mask the same as the Data; AND will set only the bits in the mask that are already set in both the Mask and the Data; OR will set bits that are set in either or both and XOR will set those bits that are different in the two bitmaps.

Cover will copy the mask and surround the bits that are already set with extra bits. The source bitmap should not have any pixels set around each edge as these will be cleared in the source so that it can be covered correctly. This is useful for producing the first attempt at an Icon's mask from its Data bitmap; this is another command that is best understood by experiment. As usual you will be warned about the area that will be destroyed before proceeding if you are not in Expert Mode. If you delete something unintentionally you can always use Cancel to revert to the Icon before you entered the Icon editor.

Zoom causes the current selected bitmap, Data or Mask, to take up the whole of the main display; this is intended for editing large Icons where each pixel is very small in the main display. Click on Zoom again to return to the normal display.

Visually a finished Icon has three components; the Bitmap part, the String (ICON in the example above) and the Character (A in the example above). Every object of type Icon has an overall size just like any other GEM object; this is normally bigger than the Bitmap itself. The three components may each be positioned independently relative to the top left corner of the object. In the example, the Bitmap is to the left of the box and the Text near the bottom in the middle. Strangely the single Character's position is actually relative to the bitmap not the main object. Also GEM will draw the Bitmap and String even if they are outside the object's box.



To edit the text of the Icon's String, double-click on the text in the Actual Size display; this will bring up the usual text name box so that you can enter the String and also set the Name of the Icon object. The Icon Text may also be moved in the normal manner. Editing the Icon's Character works in a similar way and you may also move the Bitmap itself within the nominal box represented by co-ordinates of the object. The object's co-ordinates are changed as usual in the Object Level window.

Frequently Icons do not need either or both of the String and Character attributes; you can just set these to be blank. So that you can edit such text, on entry to the Icon Editor, blank strings are represented as _ _ _ _ and blank characters as _ . As a result of this and the fact that the actual display for icons is simulated (so that WERCS knows accurately where the components are) the Actual Display is not quite the same as the GEM display in the main Object Level window or when the Icon is displayed by your program.

We will now describe the menus in detail.

## File Menu

The File menu is used to manipulate which file you are editing. Initially you are editing a blank file, shown within a window labelled Untitled.

### New

To starting editing a new empty file, click on the New item from the File menu.

### Loading

To load an existing resource file click on Load and select the appropriate file from the File Selector. An alert box saying `.HRD file not found` will appear if this file is missing - you will still be able to edit your file although any names previously attached to trees or objects will be lost.

Another way of loading a file is to set up the GEM Desktop to load WERCS when you click on .RSC or .HRD files, using Install Application. Similarly you may invoke WERCS from a CLI (such as Craft) in which case the file extension is not required.

In addition, if you launch WERCS from within the HiSoft BASIC editor (via the Tools menu), the resource file that is connected with the BASIC program that is being edited will be loaded automatically.

If you wish to load a resource file created with another resource editor you may like to convert the other editor's equivalent of the HRD file into a true .HRD file, in order to preserve the names of your items, using the conversion utility WCONVERT.

### Importing Images

Images and Icons converted using the WIMAGE utility can be imported using the Import Image item on the File menu. You will be presented with the File Selector to enter the file to import. This will copy the object to the Clipboard, subsequently selecting Paste from the Edit menu will place it in your file.

This command actually copies the second object from the first tree in the file to the Clipboard.

### Saving

Clicking on Save As will present you with the standard file selector and you can choose a filename for the current file. Clicking Save saves the file, without pause, under its original name - if the file was Untitled then Save will do a Save As.

The filename you enter into the File Selector for Save As need not have any extension - suitable extensions will be added by WERCS. In addition to a .RSC file, an .HRD (for HiSoft Resource Definition) file is also saved. This is a special file which contains such details as the names you have selected for the contents of that file. The .HRD file format is described in detail in the *Technical Reference* manual.

### Save Prefs

The default values for various options are read in from a file called WERCS.INF. This is searched for on the standard GEM path - see *Setting the Path* in the *Editor* section for more detail.

To change the defaults use the Save Prefs item on the File menu.



### Quit

To leave WERCS, click on Quit. If you have changed the file you are editing you will be given the opportunity to save or lose your modifications. You can also use the *Close* box on the tree window to achieve the same effect. If you launched WERCS from within the HiSoft BASIC editor, you will be returned to the editor when you leave WERCS.

## Flags Menu

This menu contains the various attributes that are part of the `ob_state` and `ob_flags` fields of object tree items. A selected item is shown ticked. The corresponding standard GEM names for the fields are as follows :

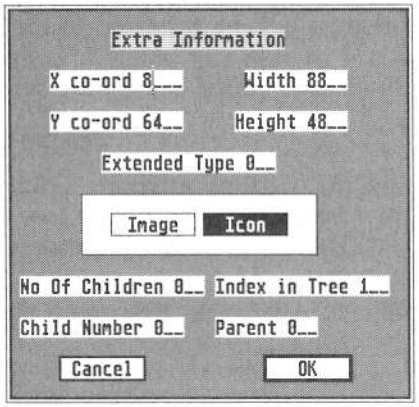
| | |
|---|---|
| Selectable | SELECTABLE |
| Default | DEFAULT |
| Exit | EXIT |
| Editable | EDITABLE |
| Radio Button | RBUTTON |
| Touch Exit | TOUCHEXIT |
| Hide | HIDETREE |
| Selected | SELECTED |
| Crossed | CROSSED |
| Checked | CHECKED |
| Disabled | DISABLED |
| Outlined | OUTLINED |
| Shadowed | SHADOWED |

### Un Hide Children

The item Un Hide children will clear the HIDETREE bit for any immediate children of the selected object so that they become visible and you may then select them once more.

## Extras

This displays a dialog box similar to that shown below which allows direct access to the object's internal structure and thus care should be taken when using this command.




The X, Y, Width and Height items are relative to the object's parent in pixels. These may be modified; use this with care as you can easily make objects move outside their parents.

The Extended Type is the most significant byte of the object word. This is ignored by the AES but may be used for your own purposes.

No Of Children is the number of first generation children that an object has. It does not include 'grand-children'.

Index in tree is the object number relative to the root object of the tree.

Child number is 0 for the first child of its parent, 1 for the second and so on. This field may be changed, in which case the objects between the old and new positions will change position in the tree. The easiest way to place the children of a particular parent in a particular order is to select the first child and make this child 0 then select the second child and make this child 1, etc.

Objects may also be re-ordered using Sort from the Misc menu.

Parent gives the Index in tree number of the object's parent.



The buttons in the box (Image and Icon in the above example) tell you what type the selected object is and let you change the object's type. Image may be changed to Icon, Box to IBox, String to Button, and Text, FText, BoxText and FBoxText interchanged.

Changing Icons into Images loses the mask and string items of the Icon so you are prompted for confirmation unless you are in Expert Mode.

## Fill Menu

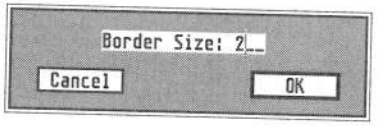
The Fill menu lets you change the fill pattern of the object, the colour of the fill and whether it is *opaque* or *transparent*. Opaque means that text will be displayed with a white background whereas transparent means that the fill pattern and fill will show 'behind' the text.

The fill pattern and colour are only applicable to Box, BoxChar, BoxText and FBoxText objects. The transparent/opaque setting is only applicable to BoxText, Text, FBoxText and FText objects.

The Fill menu is also used to set the background colours of icons.

## Border Menu

Lets you change the colour and size of the border for an object. Clicking on the Size item brings up a box as below:




The Border Size is specified in pixels as appropriate. A negative size means the border is drawn inside the box; a positive number means it is drawn outside. This is only normally useful with Box, BoxChar, BoxText, FBoxText and IBox objects.

If set for FText or Text objects, the border affects the size of the box that is drawn when the object is selected thus increasing or decreasing the visual size of the object.

## Text Menu

The Text menu lets you change the justification, colour, and size of the text object types: BoxChar, BoxText, FBoxText, Text and FText. The actual text is changed by double-clicking on the object.

## Clipboard

The clipboard is a special area of memory which can contain trees or objects. It is ideal for moving or copying items between different areas of a resource file, or between different resource files. All clipboard commands can be found on the Edit menu.

### Cut

Cut will copy the currently-selected object to the clipboard with its children and removes the current object from the tree. If the object has children you will be asked if you wish to delete them as well. If you choose not to delete the children they will become the children of the deleted object's parent. The object may then be pasted somewhere else.

### Paste

Changes the mouse form to a pointing finger and waits for you to left-click. This will place a copy of the object at that position. To cancel this, click on Cancel on the Edit menu.

### Copy

Copy copies the current selection to the clipboard and leaves it in place.

### Cancel

Cancel is used to cancel the selection of a menu when the mouse has changed to a non-pointer form. For example, if you click on String from the Object menu and decide that you do not want a new string after all, click on Cancel.

From within the Image/Icon editor, Cancel will cancel all the changes you have made since you entered the Image/Icon editor. You are prompted with a dialog box first to ensure that this is really what you wish to do.



### Abandon Edit

This allows you to abort the object-level editing that you are currently performing. All changes made since you chose to edit the current tree will be lost. It's ideal if you have made a major mistake in editing a particular tree.

### Delete

Delete works like Cut but leaves the contents of the clipboard intact.


To copy just some of the objects of a parent, create a new Box using the Object menu and make this cover the objects you wish to copy. Then select Copy and use Delete (*not* Cut) to delete your temporary Box, but not its children. Now use Paste to place the copy of the objects where you need them and then Delete to remove the outer Box again. This sounds more complicated than it is in practice.

## Misc Menu

### Auto Size

With Auto Size enabled, every time you change the text of an object the size of the object's box will change to just surround it; thus if you make the text of a Button longer it will make the Button bigger; shortening the string will make the box smaller. If you switch Auto Size off, the Button would stay the same size and the new text would not necessarily fit in the existing box.

### Auto Naming

If Auto Naming is enabled (shown by a tick) then objects are automatically given a Name as if the Same button in the Text dialog box had been clicked. The Name is based on the tree's prefix, if any, and the Text of the item.

### Auto Snap

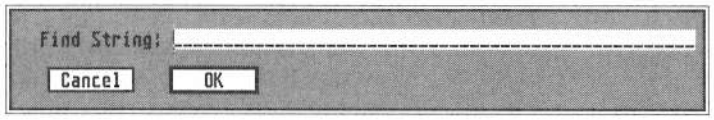
If this item is selected from the Misc menu then every item that you move or size will be *snapped* to the nearest character boundary. This is useful to make sure that items line up and will appear the same in different screen resolutions.

## Half Char Snap

If this item is selected from the Misc menu then objects will snap to the nearest half-character boundary, in a similar way to character snap. However if you are designing a resource file to run in more than one resolution then objects will not necessarily come out the same, as half a character in one resolution may be either a whole character or a quarter of a character in another resolution.

## Find Text

This enables you to find occurrences of a particular string within the text fields of the objects. You are presented with a dialog box, as below,




For example, if you have a number of menus and cannot remember which menu contained the item Stop you could use this command. The appropriate tree is opened and the object containing the string is selected.

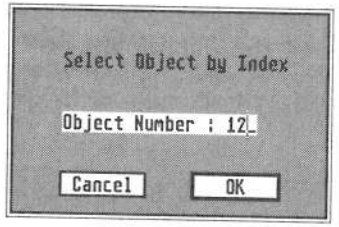
## Find Name

This item searches for a particular named object within the file, opens the appropriate tree and selects the object. The box presented looks like this:



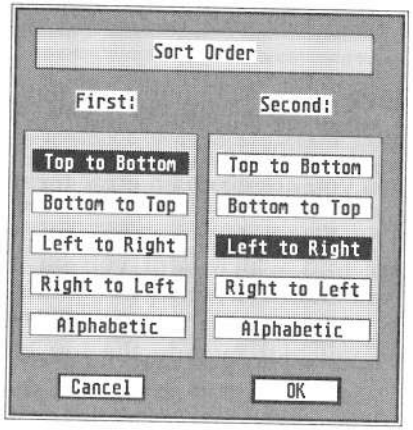

## Number Select

This allows you to select an object given its *object number* in the current tree; this can be useful if you have an object outside its parent.




## Sort

This enables you to sort the children of a particular object according to various possible criteria that are selected from the dialog box:




Top to Bottom and Bottom to Top will sort the objects according to their y position on the screen whilst Left to Right and Right to Left will sort them according to their x position on the screen. There are two priorities for the sort, First and Second. Note that the sort *does not* affect the objects' positions on the screen, it affects their order *in memory* and *within* the tree.

The default is First, Top to Bottom and Second, Left to Right.

So, say we have 6 objects (names obja to objf in any order in the tree and in memory) with screen representations as follows:

objd obja obje

objb objf objc

and then we sort using the default options. The objects will be sorted so that their order in memory and in the tree is objd, obja, obje, objb, objf, objc. Note that the sort will *not* affect the screen representations.

Alphabetic means that the *strings* of the objects are compared rather than their screen positions. Sorting alphabetically does *not* mean that the objects will change position on screen only that their position in the object tree and in memory may change.

Remember to select the parent of the objects that you wish to sort before you click on Sort. You can use Alt-clicking to select the parent of a given object.

### Test

Test lets you test out a Form, Menu or Alert Box. In order to test a Form it must have an object (such as a Button) with the EXIT and SELECTABLE flags set, or alternatively with the TOUCHEXIT flag set. If it does not then you are given an error message.

When you click on an Exit Button (or click on an item if testing a Menu) then you are told which item you have selected and its name, if any. You can choose whether to continue testing the tree, or return to WERCS. If you double-click on a TOUCHEXIT item then the value displayed will not include the top bit, as returned by `form_do`.

### Expert level

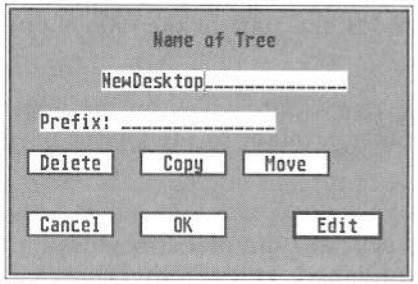
If this is enabled (shown by a tick) then all warnings to do with tree re-organising are suppressed: for example, when an object is given a new parent, or commands that effect the entire data or mask bitmaps in the icon editor. This also includes the warning about losing information when changing an Icon to an Image using Extras and when using the Abandon Edit command.



## Tree Level Editing

### Forms

Forms are the most common type of tree in resource files; they are normally used for dialog boxes, but can also be used for replacement desktops, to change the pattern of the background in a GEM program or to add icons to it. When you create a new Form or single-click on an existing one in the file window, you are presented with a dialog box like the one below:




The name of the tree may be changed. WERCS will check to ensure that it is a valid name according to the current selection of Language, with the correct case, and that it is not a duplicate of a current name. If the other item with this duplicate name is an object you can use the Find Name command to select it and then change its name.

Remember that in Mixed case Ok and OK are different names but if you have selected Lower or Upper case then they are not.

Pressing the Return key or clicking on the Edit button will then let you edit the objects within the tree.

Cancel will not add a new tree to the file and will disregard any changes to the tree name that you have made.

To add more than one tree without editing them immediately, click on the OK button - this lets you set up a number of trees without entering the objects.

To re-order the trees in a file, click on the Move button. This will change the mouse form to a pointing finger; you should then click on the tree that you wish to place immediately *after* the current form. Owing to the structure of resource files, when the file is reloaded, the Menus and Forms will be first, followed by the Free Strings and Alerts, followed by the Free Images.

To delete or copy an entire tree, click on the appropriate button. To paste a tree from the clipboard into your file, click on Paste from the Edit menu.

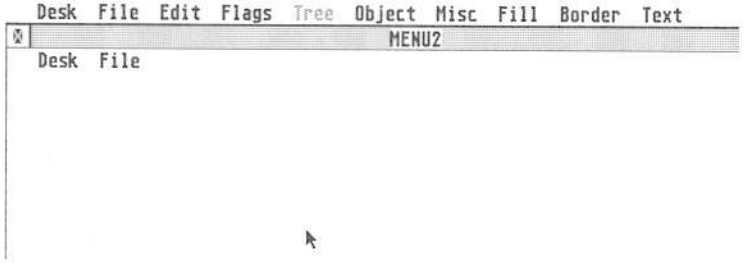
This box also lets you set up the Prefix for this particular tree. This is used to provide the start of the names of objects if you use Auto Naming.

To edit an existing tree, double-click on the appropriate tree icon. You will then be taken straight to the Object Level.

## Menus

Menus are a very special type of Form which must conform to a number of un-published rules, otherwise GEM will behave strangely. Fortunately, when using WERCS, you don't have to worry about these rules as WERCS will cope with them for you.

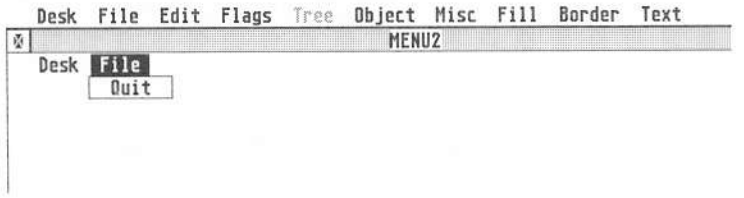
When you ask for a new menu you will see a screen similar to that below:




Normally Menus consist of Titles (which are displayed along the top of the screen) and Strings (which are displayed in the pull-down menus themselves). To add a new Title, click on Title from the Object menu and then click in the menu bar where you would like it. The other Titles (and their menus) will be moved if required.



To add items to a given Title, first click on the Title itself; this will cause the appropriate Menu to appear, for example:




You can then insert objects in the usual manner, normally Strings, and objects below the new object will be moved down. You should ensure that the mouse pointer is at the left edge of the box when inserting objects; otherwise you will leave a 'hole' to the left of it.

You can use types other than Strings in Menus if you wish; we used WERCS to produce its own resource file, for example. You can also change the flags and states of items just as if you were editing a Form. For those objects that have them, the items on the Fill, Border and Text menus can be used.

If you want your menu to work in more than one resolution don't use Icons or Images or you will find that there will either be gaps between them or they will overlap. This is because the width and height of these objects are a different number of character cells in different resolutions. This can be avoided by adjusting the object once it is loaded so that there are no gaps between icons.

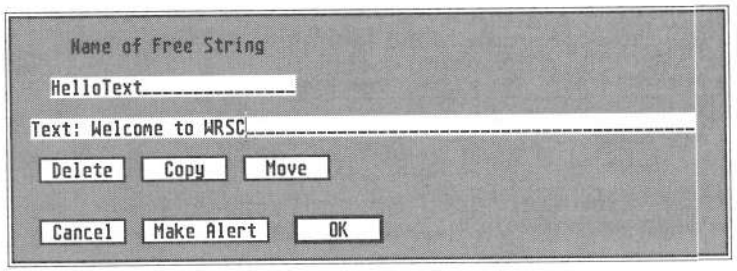
The boxes surrounding menus must not be take up more than one quarter of the screen, otherwise the system may crash. Be especially careful, when designing menus for use in Low Resolution.

The Tree Name box can be used in the same way as for Forms.

## Free Strings

A Free String is a string of characters that is not connected with any particular tree. They can be used to facilitate foreign-language versions of software, for example.

The Name and Text of the Free String can be modified in the same way as any other type of string in WERCS so that you can use \ \ to enter control and graphics characters for example.




To edit an existing Free String, it is only necessary to single- or double-click to bring up the box shown above.

The Delete, Copy, Move and OK buttons work in the same way as with Forms, detailed earlier.

Make Alert can be used to turn a Free String into an Alert. You should ensure that the String conforms to the rules for Alerts, as described below.

## Alerts

Alert Boxes are actually stored as Free Strings (see above) but are passed to the AES `form_alert` call to display an alert box.

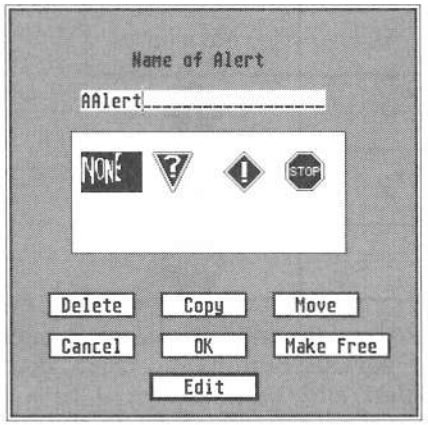
There are two types of restrictions as to the contents of Alerts; the first type is those restrictions documented by Atari (to keep down the amount of memory used by the AES when displaying them) and the second type of restriction is caused by bugs in the first release of the operating system ROMs.

As officially documented, each line in an alert box must be no more than 30 characters and there is a maximum of 5 lines. Each Button must be no more than 20 characters each and there is a maximum of three Buttons. Strings and Buttons may not contain ] or | characters. ROMs prior to 1.2 do not check for the infringement of these rules and failing to adhere to them will corrupt certain areas of the AES workspace! WERCS rigidly enforces these rules when converting the tree representation back to a string.



The release 1.0 of the ST ROMs contained various bugs, including long buttons/short text problems. Before releasing a commercial program ensure you have checked all your Alerts on a 1.0 ROM machine. Subsequent ROM releases (1.2 a.k.a. Blitter TOS, 1.4 a.k.a. Rainbow TOS and 1.6 a.k.a. STE TOS) have these problems corrected. These difficulties with the early ROMs have not been documented and so WERCS cannot check reliably for them. If you have later ROMs you have the flexibility to use some Alerts that it is not possible to use with the earlier ROMs.

The dialog box that you are presented with, when you single-click on an Alert Box in the file window, looks like this:




Which icon will appear in the Alert Box is controlled by clicking on the appropriate icon in the tree display as above.

The Delete, Copy, Cancel, OK and Move buttons work in a similar way to those on the Form Tree Name dialog box.

Clicking on Make String turns this item into a Free String rather than an Alert.

Clicking on Edit (or double-clicking on the the icon from the tree level display) will open an Object Level window that looks similar to that opened when you are editing a Form.

You should only add Strings and Buttons to the Form and these will be re-positioned automatically by WERCS. You can edit the Text in the normal way and also delete, copy and paste objects. Modifying the flags and states of the parts of an object will not affect the final Alert Box.

Re-ordering the Buttons in Alert Box is achieved by dragging a Button. If you drag button A onto Button B then the Buttons will be re-arranged so that Button A is immediately before Button B. This is similar to the moving of Titles in Menus. If it sounds complicated, experiment and you should soon get the hang of it.

Alerts are represented as strings of the format shown below:

```
[icon][line1|line2 ... |linen][button1|button2....]
```

where `icon` is one of:

| *Value* | *Icon* |
|---|---|
| 0 | No icon |
| 1 |  |
| 2 |  |
| 3 |  |

`line1`, `line2` etc. are the various message lines and `button1`, `button2` are the various Buttons. To check that you understand this, create an Alert and then change it to a Free String and, assuming that it is small enough to fit on the screen, you will be able to inspect it.

### Free Images

A Free Image is an Image-type object that is not connected with any particular tree. When you use `rsrc_gaddr` you get the address of a BITBLK rather than an object.

The Tree Name dialog box for Free Images works in the same way as that for Forms except that clicking on Edit takes you straight to the Image Editor as for Image objects.



## Keyboard Shortcut Summary

The following table gives the keyboard shortcuts when the Alt, Ctrl or Shift keys are held down.

| *Key* | *Alt* | *Ctrl* | *Shift* |
|---|---|---|---|
| A | Abandon Edit | Border Size | Alert |
| B | | Shadowed | Box |
| C | Copy | Crossed | BoxChar |
| D | | Default | Form |
| E | Extras | Editable | Button |
| F | Find Text | | FText |
| G | Find Name | Disabled | FBoxText |
| H | Selected Number | Delete | |
| I | Import Image | Small text | IBox |
| J | | Large Text | Icon |
| K | Expert | Hide | Image |
| L | Load | Left | Free Image |
| M | | Centre | Menu |
| N | New | Right | |
| O | Sort | Outlined | |
| P | | Opaque | ProgDef |
| Q | Quit | Transparent | |
| R | Save As | Radio Button | Free String |
| S | Save | Selectable | String |
| T | Test | Touch Exit | Text |
| U | Auto Naming | Un Hide | BoxText |
| V | Paste | Selected | Title |
| W | Auto Size | | |
| X | Cut | Exit | |
| Y | Char Snap | Checked | |
| Z | Half Snap | | |

Note also that the Backspace key is used to delete objects, whilst the Undo key cancels an operation.

There is a rationale behind the choice of keyboard shortcuts to help you remember them; the Alt keys refer to commands on the File, Edit and Misc menus, Ctrl for the Flags, Fill, Border and Text menus and Shift for the Object and Tree menus.

We have attempted to make shortcuts use the initial letter of the item as far as possible; the exceptions to this are the standard clipboard shortcuts and the following:

| | |
|---|---|
| 0 | Order (Sort), |
| Ctrl-G | Greyed (Disabled), |
| Ctrl-B | Border (Shadowed). |



# MonSTB
# The Debugger

## Introduction

MonST was originally designed as a low level debugger for debugging assembly language programs but we, and many other people, have found it useful in debugging BASIC programs. The version of MonST that we supply with HiSoft BASIC, MonSTB, has been enhanced so that it 'knows' about where, in memory your program lines start and automatically loads your program and source code for you.

Together with the facilities of the original MonST, such as function labels, a full integer expression evaluator, MonSTB gives you many of the features of a high level symbolic debugger. It does not give you access to local variables, floating point numbers, functions etc. As it was originally designed as a low level debugger some knowledge of assembly language is useful when using MonSTB.

As MonSTB uses its own screen memory, the display of your program is not destroyed when you single-step or breakpoint, making it particularly useful for graphical-output programs such as GEM applications. It also uses its own screen drivers so it is possible to single-step into the operating system screen routines such as the AES or BIOS without affecting the debugger. MonSTB will also work in low resolution, thus allowing you to debug programs that run in low resolution.

If you are already an expert at using the version of MonST supplied with DevpacST version 2 then please read the next few pages on the use of MonSTB with HiSoft BASIC.

# Preparing to use MonSTB

If you are going to debug a program using MonSTB you should ensure that you have selected the correct options when compiling your program; see *Debug Options...* in the *Compiler* section above for details. Essentially you should choose Extended Symbolic debug and ensure that debugger line numbers are included.

Using the correct options will ensure that the addresses of your sub-programs/functions, the library functions that you use and the address corresponding to each line of your source code will be stored in your executable file. Don't be surprised if this causes it to increase in size dramatically! You will also need to ensure that your program leaves enough memory for the operating system so that the debugger can use this to load your source code into memory. To do this you will need to use the Leave item from the Advanced compiler option box; by default this is 4K; you should increase this by just over the size of your source code. So if your program is about 9K long a value of 14K would be appropriate. This naturally assumes that you are not already using the Keep or Leave options in your program because you are loading a resource file say.

You must then compile your program to disk since the debugger cannot debug your program directly from memory.

# Invoking MonSTB

## From the Desktop

MonSTB is supplied as a GEM program with extension .PRG; to debug a TOS application you can rename it as MONSTB.TOS or install it as a TOS program. This will ensure that the operating system will perform the same initialisation as if you were running your program without it. Once executed MonSTB will prompt you for the name of the file to load.




If you debug a TOS program with the GEM version of the debugger it will work fine but the screen display will probably be messy; however, debugging a GEM program with a TOS debugger will cause all sorts of nasty problems to occur and should be avoided.

## From the Editor

If you are using the standard editor configuration file, MonSTB can be invoked by selecting the Debug option on the Tools menu or by pressing Alt and the 2 key on the numeric key pad. Of course, the editor will need to be able to find the MONSTB.PRG file for this to work. Invoking the debugger in this way will load the .PRG file corresponding to the file being edited.The debugger will also load your source file too.

The type of initial screen mode used when invoked from the editor is determined by the GEM and TOS buttons in the Tool Configuration dialog. See the editor section for more details. The rules described above about using the wrong type of screen initialisation are also relevant here.

## From a shell

If you wish to invoke MonSTB from a command-line shell, just type

```
MonSTB test
```

if test is the program that you are debugging.

# MonSTB Dialog and Alert Boxes

MonSTB makes extensive use of dialog- and alert-boxes which are similar in concept to those used by GEM programs but have several differences. MonSTB does not use genuine GEM-type boxes in order for it to remain *robust* - that is to avoid interaction when debugging programs that themselves use GEM calls. In addition the mouse is not available within the debugger itself which makes objects like true GEM buttons impossible.

A MonSTB dialog box displays the prompt ESC to abort above the top left corner of the box together with a prompt, normally followed by a blank line with a cursor. At any time a dialog box may be aborted by pressing Esc, or data may be entered by typing. The cursor, Backspace and Del keys may be used to edit entered text in the usual way and the whole line may be deleted by pressing the Clr key - note that this is different to GEM dialog boxes which use the Esc key to delete a whole line of text. An entered line is terminated by pressing the Return key, though if the line contains errors the screen will flash and the Return key will be ignored allowing correction of the data before pressing Return again. Another difference is that dialog boxes that require more than one line of data to be entered do not allow the use of the cursor up and down keys to switch between different lines - in MonSTB the lines have to be entered in order.

A MonSTB alert box is a small box displaying a message together with the prompt [ Return ] and is normally used to inform the user of some form of error. The box will disappear on pressing the Return or Esc keys, whichever is more convenient.

## Initial Display

If you have run MonSTB without a command line you will be presented with a dialog box prompting for an executable program name. You should enter the name of the program that you wish to debug. If you omit the file's extension, MonSTB will look for a .PRG, .TTP and .TOS file in that order.


Certain features work differently or are not available when using MonSTB in low resolution. They are shown with this icon.


MonTTB has more features than MonSTB. These are shown with this icon and can be safely ignored by ST users

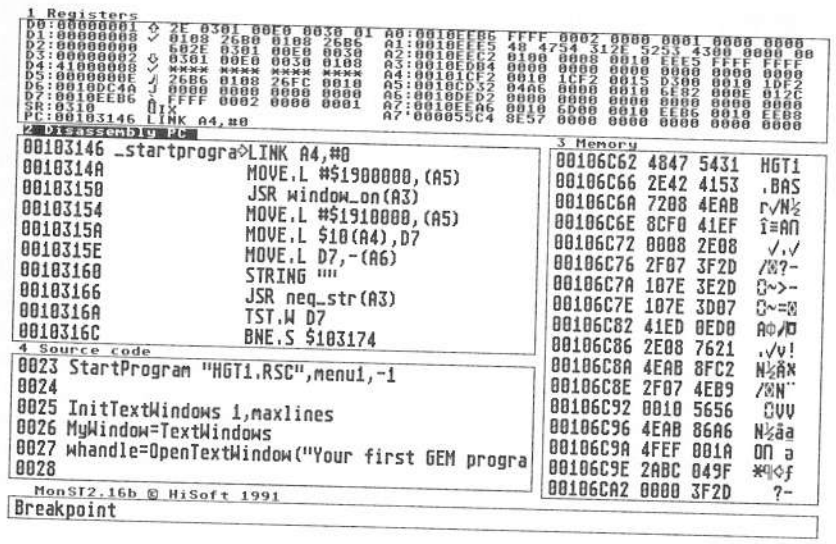


## Front Panel Display

The main display of MonSTB is via a *Front Panel* showing registers, memory and instructions. The name Front Panel stems from the type of panels that were mounted on mainframe and mini computers to provide information on the state of the machine at a particular moment, usually through the use of flashing lights. These lights represent whether or not particular flip-flops (electronic switches) within the computer are open or closed; the flip-flops that are chosen to be shown on this panel are normally those that make up the internal registers and flags of the computer thus enabling programmers and engineers to observe what the computer is doing when running a program.

So these are *hardware* front panel displays; what MonSTB provides you with is a *software* front panel - the code within MonSTB works out the state of your computer and then displays this information on the screen.

The initial MonSTB display consists of five windows, similar to those shown below. In low-resolution the arrangement of the windows is slightly different to allow efficient use of the smaller available screen space.




*The MonSTB windows*

The top window (`1 Registers`) displays the values of the machine's data and address registers, together with the memory pointed to by these registers.

The next window (`2 Disassembly PC`) is the disassembly window; this displays several lines of assembly instructions, by default based around the program counter (`PC`), shown in the title area of the window. A ⇒ sign is used to denote the current value of the PC.

Window number 3 is the memory window which displays a section of memory in word-aligned hex and ASCII.

Window number 4 is the source code window; it shows a a portion of your source code and the corresponding line numbers.

The final window at the bottom of the screen, which is un-numbered, is the smallest window and is used to display messages.

One of the most powerful features of MonSTB is its flexibility with windows - an extra window may be created, the font size can be changed, and windows may be locked to particular registers; these features are detailed later.


MonTT 'understands' about the six different screen resolutions available on the TT. If you are using a monochrome A3 screen, MonTT can display considerably more information whereas in either low-resolution the arrangement of two of the windows is slightly different to allow efficient use of the narrower screen. Whichever screen resolution you are using the windows are numbered in the same way.

If you are working with one of the colour modes you can use a different screen resolution for MonTT than that of your program (use the Ctrl-O command described in the *Technical Reference* manual). Thus, in general, it is best to use the TT medium resolution mode since this gives you the full 80 character width whilst allowing 30 rows vertically.



## Simple Window Handling

MonSTB has the concept of a *current window* - this is denoted by displaying its title in black. The current window may be changed by pressing the Tab key to cycle between them, or by pressing the Alt key together with the window number, for example Alt-2 selects the disassembly window. (AZERTY keyboard users please note - the Shift key is *not* required when using Alt to select windows). Note that the lowest window can never be made the current window - it is used solely for displaying messages.

## Command Input

MonSTB is controlled by single-key commands which creates a very fast user-interface, though this can take getting used to if you are familiar with a line-oriented command interface of another debugger. Users of HiSoft DevpacST and HiSoft debuggers on other machines should find that they are already familiar with many of the commands.

In general the Alt key is the window key - when used in conjunction with other keys it acts on the *current window*.

Commands may be entered in either upper or lower case. Those commands whose effects are potentially disastrous require the Ctrl key to be pressed in addition to a command key. The keys used were chosen to be easy to remember, wherever possible. Commands take effect immediately - there is no need to press Return - and invalid commands are simply ignored. The relevant sections of the front panel display are updated after each command so any effects can be seen immediately.

MonSTB is a powerful and sometimes complex program and we realise that it is unlikely that many users will use every single command. For this reason the remainder of the MonSTB manual is divided into two sections - the former is an introduction to the basic commands of the program, while the latter is a full reference section and is to be found in the *Technical Reference* manual. It is possible for new users and beginners to use the debugger effectively while having only read the *Overview*; don't be intimidated by the *Reference* section.

# MonSTB Overview

The most common low-level command in MonSTB is probably single-step, obtained by pressing Ctrl-Z (or Ctrl-Y or Ctrl-W if you find it more convenient). This will execute the instruction at the PC, the one shown in the `Register` window and, normally, also in the `Disassembly` window. After executing it the debugger re-displays the values of the registers and memory displayed, so you can watch the processor execute your program, step by step. Single-stepping is the best way of going through sections of code where you don't understand what is going on, but it is also the slowest - and it deals with your program only on an assembly language level, not in terms of your BASIC program. There is, of course, an answer.

A *breakpoint* is a special word placed into your program to stop it running and enter MonSTB. There are many types of breakpoint but we will restrict ourselves to the simplest for now. A breakpoint may be set by pressing Alt-B, then entering the address you wish to place the breakpoint. You can enter an address in MonSTB as a symbol, as a hexadecimal line number preceded by #, hex (the default base), a decimal line number preceded by #\ or as a complex expression. Examples of valid addresses are `MainLoop`, `foo`, `#10`, `#\123`, `10+mydata`. If you type in an invalid address the screen will flash and allow you to correct the expression.

Having set a breakpoint you need some way of letting your program actually run, and Ctrl-R will do this. If will execute your program using the registers displayed and starting from the PC. MonSTB will be re-entered if a breakpoint has been hit, or if a processor exception occurs.

MonSTB uses its own screen display which is independent from your programs. If you press the V key you will see your current programs display, pressing another key switches you back to MonSTB. This allows you to debug programs without disturbing their output at all.

Any window may be zoomed to the full screen size by pressing Alt-Z. To return to the main display press Alt-Z or the Esc key. The Esc key is also the best way of getting out of anything you may have invoked by accident. The Zoom command, like all Alt-commands, works on the *current window* which you can change by pressing Tab. You can dump the current window to your printer by pressing Alt-P.



To change the address from which a window displays its data, press Alt-A, then enter the new address. Note that the disassembly window will always re-display from the PC after you single-step, because it is *locked* to the PC. The locking of windows is detailed in the Reference section.

To quit MonSTB press Ctrl-C. Strange as it may sound this will not always work - what Ctrl-C does is terminate the *current program*, which may be MonSTB or, more likely, the program you are debugging., in which case MonSTB will still be in control. You know when you have terminated the program under investigation because it will say so in the lower window. Once your program has been terminated, pressing Ctrl-C will terminate MonSTB.

We hope this quick overview has given you a good idea of the most common features of MonSTB to let you get on with the complex process of debugging programs. When you feel more confident you should try and read the *Reference* section in the *Technical Reference* manual, probably best taken, like all medicine, in small doses.

# Chapter 4
# Concepts

This chapter describes the technical details of the HiSoft BASIC language, together with some of its more advanced features. It is intended for users who already have a good understanding of the BASIC language and want to get to grips with HiSoft BASIC quickly.

If you are new to BASIC please read the *Tutorials* chapter first.

## Character Set

HiSoft BASIC uses plain ASCII characters in its input files. The following characters have special meanings:

| | |
|---|---|
| a-z, A-Z | The letters, which are used in reserved words and the user's variable names, labels and sub-program names. Lower and upper case are treated as the same in variable and reserved word definitions so that `THEN`, `then` and `Then` are all the same reserved word. |
| E, e, D, d | These are also used for exponents in numbers. |
| 0-9 | The digits, which are used in numbers and can also be used in names as long as they are not the first character. |
| . | The full stop or period, which is used as the decimal point in numbers and can also be used in names as long as it is not the first character. |
| % | The percentage sign, which is used to indicate that a variable is a 16 bit integer i.e. whose values must be in the range -32768 to 32767. |

| | |
|---|---|
| & | The ampersand, which is used to indicate that variables are long integers i.e. whose values must be in the range -2^31 to 2^31-1. Also used to introduce hexadecimal, octal and binary constants. |
| ! | The exclamation mark, which is used to indicate that a variable is a single-precision floating point number. |
| # | The hash or number sign, which is used to indicate that a variable is a double precision floating point number and also used to indicate that certain input/output operations are to be directed to channels rather than the screen (e.g. `PRINT #`). |
| $ | Used to indicate string variables. |
| _ | The underline character, which can be used in variables after the first character assuming the underline (U) flag has not been turned off. If it appears at the start of a symbol or the underline (U) has been disabled then it indicates the rest of the line is to be ignored and that the following line is to be considered part of the current one. |
| " | The quotation mark or double quote which is used to delimit string literals. |
| ' | The apostrophe or single quote which is used to indicate that the rest of this line is to be regarded as a comment. |
| ( ) | The parentheses or round brackets, which are used to enclose function arguments, over-ride the priority of operators and indicate arrays. |
| + - * / | The basic arithmetic operators. |
| = | The assignment operator and equality operator. |
| < > | Less than and greater than comparison operators - also used as parts of the shift operators. |



| | |
|---|---|
| ^ | Exponentiation operator. |
| \ | The back-slash character, which is used as the integer division operator. |
| , | Comma. |
| ; | Semi-colon. |
| ? | Used as an abbreviation for `PRINT`. |
| Ctrl -Z | If ASCII Ctrl-Z (`chr$(26)`) is found in a file then it is treated as end-of-file. |

Other characters with ASCII values lower than 32 are treated as white space, and ignored so you may, for example, include form-feed (`chr$(12)`) characters to give a new page on your printer when listed.

Other characters may be used in strings, but otherwise will generate a warning and will be ignored.

# Program lines and labels

Program lines consist of an optional line number or label, one or more statements separated by colons and an optional comment, which starts with an apostrophe or single quote.

Line numbers may be any number between 1 and 65529 inclusive. (65529 may seem a strange number, it is the maximum allowed by Microsoft BASIC).

Line labels consist of any valid variable name that is not used as a variable or a sub-program and labels are followed by a colon. There is no limit to the number of characters in a line label but lower- and upper-case letters are treated as the same and the characters must not be a reserved word. Thus the following line labels are allowed:

```
Label9999:

A.very.long.label.that.causes.problems.to.other.BASICs:

Hello:
```

The following line labels are the same:

```
Start:
START:
start:
```

Line numbers and labels may be preceded by white space. White space is not required after the last digit or colon. Line numbers may not contain spaces.

*Note:* For compatibility with other BASICs we recommend that you do not use full stops (`.`) or underlines ( `_` ) in labels and keep them to less than 40 characters.

Line number `0` is not allowed because it would be confused when using `ON ERROR GOTO 0` which does not mean go to line `0` if an error occurs.

In general we do not recommend the use of line numbers since line labels are much more readable. The exception to this is when using `ERL` when the use of line numbers is useful since otherwise you must change the line numbers in your program each time you insert or delete lines in your program.

Most of the time you do not need to use line numbers or line labels because HiSoft BASIC has such a rich set of structured statements (much better than Pascal and even more flexible than C and Modula 2).

You may have many statements per line provided each is separated by a colon.

HiSoft BASIC has an extension to call sub-programs without the `CALL` keyword. However you cannot do this if the sub-program has no parameters and is the first statement of a multi-statement line. For example, if you have:

```
SUB john STATIC
PRINT "John";
END SUB
```

then

```
john
PRINT " Smith"
```



will print

```
John Smith
```

as will

```
call john: print " Smith"
```

However

```
john: print " Smith"
```

is wrong because the `john:` could be a label definition; normally the compiler will warn you on the parsing phase if you make this error. This problem does not apply if the sub-program has a parameter, for example:

```
SUB john(para$) static
PRINT "John ";para$;
END SUB

john "David": print " Smith"
```

is fine because it cannot be mistaken for a label.

If an apostrophe or single quote (') appears on a line then the rest of the line is treated as comment and ignored. The only exception to this is `DATA` statements which treat the apostrophe as part of the data. If you want a comment on such a line precede it with a colon; this will terminate the `DATA` statement.

Program lines may be any length theoretically, but it is generally a good idea to keep them less than 80 characters so that the whole line is displayed at once. If you need a line that is significantly longer than this then the chances are that the line is more complicated than it should be as far as ease of understanding is concerned. The exception to this is the `FIELD` statement where, for large records, you need many more than 80 characters.

To get around this the underline character (_) may be used to cause lines to be continued on the next physical line. Anything after the underline is ignored.

For example:

```
FIELD #3,20 AS name$, _ ' surname only
   5 AS initials$ ,_
   50 AS street$, _' include name or number here
   20 AS town$, _
   20 AS county$ ,_ ' or state if applicable
   20 AS country$
```

which would be much more readable than the one-line equivalent where you would also have to leave out the comments.

Normally HiSoft BASIC allows underlines in variable names, unlike traditional BASICs. Underlines are treated as continuation characters if they are not part of a variable name. If you are porting programs that have continuation characters immediately after identifiers or reserved words then use the U- option or click on the appropriate box in the Compile... dialog box.

For example:

```
IF x THEN_
  PRINT "hello"
```

will be accepted in some BASICs as a one-line IF statement (no need for an END IF). Without the U- flag HiSoft BASIC will give an error because it thinks you are using a variable called THEN_. To solve this use U+ or insert a space in front of the _ character.

If you use U- and inadvertently use an identifier containing an underline you may get a very strange error message because the compiler has ignored the rest of the line.

# Data Types

There are five types of data in HiSoft BASIC:

## Strings

A string is a sequence of characters that may be up to 16 megabytes long assuming you have enough memory. Strings may contain any character with a value of 0 to 255 inclusive.



## Integers

Integers are numeric and consist of the whole numbers from -32768 to 32767.

## Long Integers

Long integers are numeric and consist of the whole numbers between -2147483648 and 2147483647.

## Single precision numbers

Single-precision numbers have approximately seven digits of precision and a range of 5.4E-20 to 9.2E-18 for positive values and -2.7E-20 to -9.2E18 for negative values.

## Double precision numbers

Double precision numbers have approximately 16 digits of precision and a range of 4.9E-324 to 1.8E308 for positive numbers and -4.9E-324 to -1.8E308 for negative numbers. There is loss of precision with numbers of magnitude less than 2.2E-308.

# Constants

Constants are values which do not change during program execution. Constants may be of all 5 types.

A string constant is a sequence of ASCII characters enclosed in double quotes ("). These can be any character between ASCII values 32 (space) and 255. To obtain a double-quote in a string repeat it, so that, for example, the string consisting of one double quote character is """". The first is the start of the string, the second and third form the character itself and the last is the closing quote.

Numeric constants are formed in one of the following ways:

## Decimal numbers

A sequence of decimal digits followed optionally by a decimal point (`.`) and more digits and/or an exponent. An exponent consists of the letter `d`, `D`, `e` or `E` followed by a decimal integer. `E` indicates single precision and `D` indicates double precision. The number may be preceded by a minus sign as may the numeric part of the exponent.

The number before the decimal point may be omitted. The number may be followed by a type specifier (%,!,& or #).

## Hexadecimal Constants

Hexadecimal constants start with `&H` or `&h` and are followed by hexadecimal digits(`0-9`, `a-f`, `A-F`). The number may be followed by a type specifier (%,!,& or #).

Hexadecimal integer constants between `&h8000` and `&hFFFF` are taken as signed 16 bit integers. Hexadecimal long integer constants between `&h80000000` and `&hFFFFFFFF` are treated as signed 32 bit constants so that for example:

| | | |
|---|---|---|
| &h7FFF | = 32767 | integer |
| &h8000 | = -32768 | integer |
| &h8001 | = -32767 | integer |
| &hFFFF | = -1 | integer |
| &h10000 | = 65536 | long integer |
| &h7FFFFFFF | = 2147483647 | long integer |
| &h80000000 | = -2147483648 | long integer |
| &hFFFFFFFF | = -1 | long integer |
| &h100000000 | = 4294967296 | double |

If you want `&h8000` to be treated as `+32768` then follow the number with & and it will be treated as a long and thus positive e.g.

| | | |
|---|---|---|
| &h8000& | = 32768 | long integer |



## Octal Constants

Octal constants start with `&O` or `&o` or just simply `&`. and are followed by octal digits (`0-7`). The number may be followed by a type specifier (%, !, & or #). The type of an un-terminated octal constant is determined by the same rules as for hexadecimal constants (see above).

## Binary Constants

Binary constants start with `&B` or `&b` and are followed by the digits `0` or `1`.The number may be followed by a type specifier (%,!,& or #). The type of an un-terminated binary constant is determined by the same rules as for hexadecimal constants described previously.

## Character constants

These start like strings of only one character and are followed by the % character and have a value equivalent to the `ASC()` of the character. However they are generally easier to read and more efficient than the `ASC()` equivalent.

## Types of Constants

The rules regarding what type a constant is are rather complicated but in general you should find that normally the compiler does what you expect. The most common problem is that some hexadecimal constants are treated as negative. If this is a problem please see the *Hexadecimal Constants* section above. The following are in *decreasing* order of importance:

1. A terminating character is used. If the number is terminated by:

| | |
|---|---|
| % | it is taken as an integer |
| & | it is taken as a long integer |
| ! | as a single precision floating point number |
| # | as a double precision floating point number. |

2. If the number is hexadecimal or octal and lies in the following range:

| | |
|---|---|
| 0 to &hFFFF | it is taken as an integer |
| &h10000 to &hFFFFFFFF | it is taken as a long integer |
| &h100000000 upwards | it is taken as a double. |

For the rules concerning whether a constant is treated as negative see above.

3. If the number is decimal and it is not a whole number and has *more* than 6 digits in the whole number and decimal parts, then it is a double.

4. If it has an exponent of the form D or d then it is a double.

5. If it has an exponent of the form E or e then it is an integer.

6. If it has a decimal point then it is a single precision number.

7. If it is a whole number less than or equal to 32767 it is an integer.

8. If it is a whole number less than or equal to 2147483647 it is a long integer.

Examples:

| | |
|---|---|
| 1 | integer |
| 1.0 | single precision (.) |
| 1.0E0 | single precision(. and E) |
| 1.00000 | single precision |
| 1.000000 | double precision (7 digits) |
| 1.000000E0 | single precision(E overides 7 digits) |
| 1D0 | double precision (D) |
| 1D0% | integer (% over-rides D) |
| 1.0& | long integer (& overides .) |
| 1D0! | single precision (! overides D) |
| 1# | double precision. |



# Variables and Reserved Words

Variable names start with a letter and subsequent characters may be letters, digits, or full stops (.). In addition underlines (_) may be included if you haven't switched this off using the U- option. For maximum compatibility with other BASICs don't use underlines or full stops.

Lower and upper case are treated as the same in variable names and reserved words so that PRINT , Print and Print are all the same reserved word.

Variables may be terminated with a type specifier % (integer), & (long integer), ! (single precision floating point) or # (double precision floating point). If there is no type specifier then the type is determined by the current DEFtype statement for the first letter of the variable. If there have been no DEFtype statements then single precision (!) is used.

Compiler error messages specifying variable names always include the type specifier that has been assumed.

For example, the following gives the types of the respective variables:

```
DEFINT i-k
DEFSTR s
DEFDBL q-r
```

| | |
|---|---|
| i% | integer |
| i | integer i% (same as above) |
| I | (also the same as above) |
| i& | long integer (different) |
| str1 | string (same as str1$) |
| real_value1 | double (same as real_value1#) |

You can not use reserved words as the names of variables or sub-programs. The reserved words are listed in full in *Appendix A*. Reserved words and variables may be entered in upper or lower case or a mixture of both.

In general using reserved words with type specifiers should be avoided for compatibility reasons.

GO is not a reserved word. However if GO is followed by TO or SUB then it is made into GOTO or GOSUB respectively; so you can have white space between GO and TO and it will still be treated as GOTO. Thus you can use GO as a variable name if you like but some strange things can happen, such as

```
FOR i=go TO from STEP 2
```

is misunderstood because the compiler considers this to be

```
FOR I = GOTO  from  STEP 2
```

Variables must not start with FN because they would be treated as function names. The same rules for determining the type of a variable are used to determine the types of functions. The following are FN function names:

```
FNtest
FNsine&
FNget.one.character
```

Sub-program names must not have a type specifier because they do not have a type. You may use the same name for a variable and a sub-program although the type specifier of the variable must be used explicitly. However, this can be *very* confusing. For example

```
SUB john
   john%=42
END SUB

john%=52 : john
```

# Arrays

Array names follow the same rules as for variables and their types are determined in the same way. You may use the same name for an array and an ordinary variable. Normally an array name is followed by an open parenthesis, except in the ERASE statement and the UBOUND and LBOUND functions when this is assumed automatically.



Arrays are tables of values each of the same type. Normally the number of elements in an array and the number of dimensions is specified with a DIM statement. There are no restrictions on the size of arrays other than available memory and subscripts may be long integer values if applicable. The maximum number of dimensions for an array is 31 (which would take up a minimum of 4 gigabytes of memory if each index had more than one element).

If the DIM statement is not used, the maximum subscript is assumed to be 10. If you have switched off array checks using compiler option A- then you *must* use the DIM statement. To check whether you have inadvertently auto-dimensioned an array then you should use the Array Checks Warnings option - this will give a run-time error when an array is used before it has been DIMensioned.

The minimum value of subscripts is 0 unless an OPTION BASE statement is used. When referenced, the element of the array to be accessed is specified by one or more expressions inside parentheses and separated by commas. The expressions may be of any numeric type although single and double precision real values will be converted to long integers.

For example, given:

```
DIM A(30), B$(table_entries,4), table&(100000)
DIM t%(fred*fred),c(n,n,n)
```

then the following are valid array references:

```
A(i)
B$(j*3,2)
table&(i&)
t%(k-1)
c(i,j,k)
```

By default HiSoft BASIC arrays are *static* i.e. you cannot re-dimension them; this can be changed through the use of the REM $dynamic compiler option - for more information on this and other features of arrays see the *Advanced Arrays* section later in this chapter.

# Operators

Expressions are made up of constants, variables, array variables, function calls and operators. The order of priority is listed below with the highest priority first:

1. Exponentiation ( to the power of ) (^)
2. Unary Minus (-)
3. Multiplication (*) and Floating Point Division (/)
4. Integer Division (\)
5. Modulus ( MOD)
6. Addition (+) and Subtraction (-)
7. Shift left (<<) and Shift right (>>)
8. Comparisons (=,<>,>,<,>=,<=, ==)
9. NOT
10. AND
11. OR and XOR (exclusive or)
12. EQV
13. IMP

The only exception to this is that `x^-y` is evaluated as `x^(-y)`.

To change the order of evaluation use parentheses (round brackets).

The guiding principle for the precision used when evaluating expressions is that the minimum precision is used that will ensure that accuracy is not lost.

The exponentiation operator (^) always has its operands converted to either single or double precision floating point and returns a result of the same type. Single precision is used if the operands are either integer or single precision. If either operand is a long integer or is double precision then it is evaluated in double precision for accuracy. See *2-2* below. This operator uses logarithms to give its result and as such is slow and inaccurate if the second operand is a small integer.



The multiplication, addition, subtraction and unary minus operators may have operands of any numeric type with the following table giving the result of the expressions:

| | **integer** | **long** | **single** | **double** |
|---|---|---|---|---|
| **integer** | integer | long | single | double |
| **long** | long | long | double | double |
| **single** | single | double | single | double |
| **double** | double | double | double | double |

*2-1 Type conversion for most operators*

Addition may be also used for strings when it means concatenation so that, for example:

```
"ABC" +"DEF" ="ABCDEF"
```

Floating point division operands are always converted to single or double precision floating point numbers, the following table gives the result of the expression:

| | **integer** | **long** | **single** | **double** |
|---|---|---|---|---|
| **integer** | single | double | single | double |
| **long** | double | double | double | double |
| **single** | single | double | single | double |
| **double** | double | double | double | double |

*2-2 Type conversion for division operators*

The integer division operator \ uses long integer (32-bit) arithmetic *unless* both operands are integers in which case integer (16-bit) arithmetic is used.

The comparison operators always return an integer value of -1 for true and 0 for false. The comparison is evaluated using the type given in *2-1* above for numeric types. Strings may also be compared.

The comparison operators are

= equality
<> inequality
> greater than
< less than
>= greater or equals
<= less than or equals
== almost equals (two equals signs)

The 'almost-equals' operator is a HiSoft BASIC extension for single or double precision floating point comparisons and it is defined as follows:

```
x==y
```

calculates

```
ABS(x-y) <= ABS(y * 1E-6)
```

Thus == can be used to check for near equality even if a small number of rounding errors have been introduced. For integers and long integers the comparison is the same as equals and for strings the comparison is the same as equals except that lower case letters are treated as equal to their uppercase counterparts. For example:

| | |
|---|---|
| `2.0==2.0` | is true |
| `2.0==1.999999` | is true |
| `2.0==1.99999` | is false |

A string is considered less than another if its first character that differs is lower in the ASCII set than the corresponding character in the first string. If the strings are the same until one string is exhausted then the shorter string is less. All the following examples are true:

| | |
|---|---|
| `Fred"<"Hello"` | because `"F"<"H"` |
| `"Frederick"<"Hello"` | because `"F"<"H"` |
| `"fred">"Hello"` | because `"f">"H"`. The lower case letters come after the upper. |
| `"Frederick">"Fred"` | because `"Frederick"` is longer |

The logical shift operators, <<,>> shift their first operand left or right respectively by the number of bits given as their second operand. The shift operations are unsigned and the first operand should be of either integer type - if you use a singlle or double this will bve converted to a long integer. The resulting type is the same as the first operand after any conversion.

Note that i<<1 is the same as i*2 except when would occur and that i>>1 is the same as i\2 except when i is negative.



For example:

| | | |
|---|---|---|
| `1<<8` | `&h100` | 256 |
| `(-1)<<8` | `&hFF00` | -256 |
| `1<<15` | `&h8000` | -32768 |
| `&h1234<<4` | `&h2340` | 9024 |
| `&h12345678<<4` | `&h23456780` | 591751940 |
| `&h12345678>>4` | `&h1234567` | 305419896 |

Note that the shift operator are not supported by MicroSoft BASIC on the Macintosh or PC.

All the logical operators `NOT`, `AND`, `OR`, `XOR`, `EQV` and `IMP` use long integer arithmetic (32-bit) unless both operands are integers in which case integer arithmetic is used. These operators work bitwise, with each bit affected as shown below.

| X | Y | NOT | AND | OR | XOR | IMP | EQV |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |

Although these operations work on the individual bits they have the same affect as the corresponding logical operators if you use -1 for TRUE and 0 for FALSE. Examples:

```
-1 OR -1  =-1
4 OR 3    = 7      (100 and 011 in binary)
-1 XOR 0  =-1
8 AND 4   = 0      (1000 and 100 in binary)
```

These logical operators can be particularly useful when writing routines that interface directly to the GEM AES as many of the flag parameters are based on bits.

# Sub-programs and User Defined Functions

Sub-programs and user defined functions are one of the most powerful features of many modern BASICs and HiSoft BASIC takes these ideas even further.

The idea of using a sub-program is to isolate part of the code of your program in a way that makes it easy to call and easy to ensure that it is not interfering with variables that are, logically, not to do with the sub-program's code.

The simplest definition of a sub-program is something like

```
SUB hello
   PRINT "hello"
END SUB
```

The SUB statement defines the name of the sub-program that we are defining and the END SUB indicates that we have finished.

Sub-program definitions may not contain other sub-program definitions.

The `hello` sub-program can be called using

```
call hello
```

or even just

```
hello
```

and will print the word `hello`. You can call sub-programs before or after their declarations.

So far this doesn't give us anything that you can't do with old-style BASIC GOSUB...RETURN statements. However by passing parameters to sub-programs we can make the sub-program work on different variables or values.



## Variable Parameters

Sub-programs may take two different sorts of parameters, *value* and *variable* parameters. By default parameters are variable parameters and are passed by *reference*. This means that if the sub-program changes the value of the variable parameter, its value will be modified globally. For example:

```
SUB TimesTwo(v)
   v=v*2
END SUB
```

If we call this using

```
INPUT " Enter a number";i
CALL TimesTwo(i)
PRINT i
```

and enter the number `42`, `i` will be modified and then twice this, `84`, will be printed.

The shortened form of the CALL statement above is:

```
TimesTwo i
```

Note that the brackets are not used when omitting the CALL keyword.

When using variable parameters, if you pass an *expression* rather than a variable of the required type then any modifications to the parameter are lost. In order for a variable parameter to be modified globally by the sub-program to which it is passed, the type of the passed variable *must* be the same as the type of the parameter and it must be a simple variable.

If we changed the calling code to be:

```
INPUT i#
TimesTwo i#
PRINT i#
```

then the variable `i#` would not be modified.

You can pass array elements as variable parameters; this causes the subscripting expression to be calculated before the sub-program is called e.g.

```
TimesTwo a(3)
```

would double the value of `a(3)`. However this should be avoided if you are using ERASE and REDIM APPEND/PRESERVE inside sub-programs; see the *Advanced Arrays* section in this chapter for more information.

If you want to call a sub-program that normally would modify the variable, but on this occasion you don't want this to happen, then enclose the variable name in parentheses e.g.

```
TimesTwo (i)
```

This forces the parameter to be passed by value. If you use a CALL statement instead you use the same method e.g.

```
CALL TimesTwo ((i))      'passes i by value
```

If you have more than one parameter for a sub-program they should be separated by commas in both the call and the definition.

For example:

```
SUB Multiply(i,j,k)
   k=i*j
END SUB

Multiply 2,3,i
PRINT i
```

This will print 6. Note that the `i` that is a parameter and used in the sub-program is an entirely different entity to the `i` in the main program.

## Value Parameters

Parameters may also be called by *value*, which means that the parameter will not be modified by the sub-program or function.



To indicate that a parameter is passed by value precede it in the definition with the keyword BYVAL. So the above example could be coded as:

```
SUB Multiply(BYVAL i, BYVAL j,k)
   k=i*j
END SUB

Multiply 2,3,i
PRINT i
```

Value parameters are more efficient than variable parameters and are a HiSoft BASIC extension. In most other BASICs with sub-programs you must use variable parameters and enclose them in parentheses. This works fine unless you forget the brackets, when you can modify your main program variables by mistake. In general make a parameter a BYVAL parameter unless you want to return a value.

By default, parameters to sub-programs are passed by reference, as variable parameters.

For compatibility with earlier versions of HiSoft BASIC you can use VAL instead of BYVAL; we do not recommend this for new programs as BYVAL is used by the latest MicroSoft BASICs for the PC.

## STATIC variables

In the examples so far we have only used parameters inside sub-programs. However sub-programs may have their own variables. For example

```
SUB Sum(val n, k)
STATIC count,total
   total=0
   FOR count=1 TO n
      total=total+count
   NEXT count
   k=count
END SUB

Sum 4,result
PRINT result
```

will print

```
10
```

which is 1+2+3+4. The word STATIC is used to introduce ordinary local variables. You can use commas to separate them. In fact if you omit the STATIC statement, the above will still work because STATIC is assumed by the compiler. However we recommend strongly that you use this statement together with the variable checks flag (V+). This will warn you if you mis-spell variables in sub-programs.

For example if we had typed

```
k=k+cont
```

in the example above, the compiler would complain that the variable cont was not declared.

STATIC variables are zeroed when the program starts running but are not modified between different calls to the procedure. In the example above if we called the Sum sub-program again Total would have a value of 10 so we must zero it each time.

## SHARED variables

You can also use variables from your main program inside sub-programs by using the SHARED statement. For example we could code the example above as:

```
SUB Sum(VAL n)
STATIC count,total
SHARED k
   total=0
   FOR count=1 TO n
      total=total+count
N  EXT count
   k=count
END SUB

Sum 4
PRINT k
```

This is however less flexible than the original example because it modifies only one particular variable.



Using SHARED variables with variable parameters which should be value parameters can lead to the following difficult-to-spot bug shown below:

```
SUB process(t)
SHARED token
   IF t=3 OR t=4 THEN
   .
   .
      token=5
   .
   .
      IF t=4 THEN   ' problem
   .
   .
      END IF
   END IF
```

One would naturally expect t to be 3 or 4 at the point marked problem since t was 3 or 4 in the previous IF statement. However if the sub-program process was called as

```
process token
```

then this would not be the case because the modification of token will also change t. This can be solved by enclosing t in parentheses or, even better, by making the parameter a value parameter. This problem can also occur if the variable token was modified by a sub-program that is called inside process, which is even more difficult to spot.

If you have some variables that are imported into many sub-programs and you wish to avoid having SHARED statements each time, you can use the DIM SHARED statement which causes the variable to be SHARED with every sub-program. For example, if you have

```
DIM SHARED debug_flag
```

then you can use debug_flag anywhere in your program.

## Recursion and Local variables

Sub-programs may be called recursively i.e. they may call themselves.

```
SUB Fibonacci(VAL n, r)
LOCAL temp1,temp2

   SELECT CASE n
      CASE 0: r=0
      CASE 1: r=1
      CASE REMAINDER:
         Fibonacci n-1, temp1
         Fibonacci n-2, temp2
         r=temp1*temp2
   END SELECT

END SUB

FOR i=0 TO 15
   Fibonacci i,res
   PRINT res;
NEXT i
```

This prints the first few numbers in the Fibonacci sequence, in which the *n*th term is the sum of the two previous terms with the sequence starting with 0, 1, ..... This is, in fact not the most efficient way to code this algorithm in HiSoft BASIC; the algorithm can also be improved very easily.

The above example also introduces `LOCAL` variables. These are like `STATIC` variables in that they cannot be accessed outside the sub-program. However a new variable is created for each invocation of the sub-program. This becomes important when you have recursive calls. In the example above if there was only one variable `temp1` then it would be corrupted during the second recursive call. Try it and see.

The memory for use of local scalar numeric variables is allocated on the machine stack. If you make heavy use of recursive calls with large numbers of local variables it is possible to run out of stack. Use the `R` option, see *Appendix B* in the *Technical Reference* manual and *Chapter 3* for details.



Strings may also be used as parameters and local variables in exactly the same way as numbers. The only difference is that the actual data in the strings is allocated on the heap and not on the machine stack.

## User-Defined Functions

As well as sub-programs you can also have user-defined functions.In HiSoft BASIC there are two methods of defining user-defined functions, `DEF FN` and `FUNCTION`. The latter is the prefered modern form so we will discuss this first.

### Using FUNCTION

The `FUNCTION` syntax makes user-defined functions use the same syntax as for sub-programs. User defined functions return results by assigning to a pseudo-variable with the name of the function.

For example, here's another coding of the Fibonacci example:

```
FUNCTION Fibonacci(BYVAL n)
   SELECT CASE n
      CASE 0: Fibonacci=0
      CASE 1: Fibonacci=1
      CASE REMAINDER:
         Fibonacci= Fibonacci(n-1)+Fibonacci(n-2)
   END SELECT
END FUNCTION

FOR i=0 TO 15
   PRINT Fibonacci (i);
NEXT i
```

This gives probably the neatest solution to this classic problem.

There is no restriction on the name of the function and the rules for parameters and local variables are the same as for sub-programs.

`FUNCTION`s must be declared before they are used. The normal way to this is to ensure that the `FUNCTION END FUNCTION` statements are before any calls of the function. If you wish to use a function before you define it then you can use the `DECLARE` statement. This specifies the parameters of a function in the same way as a `FUNCTION` statement but does not actually contain any code.

For example,

```
DECLARE FUNCTION Fibonacci(BYVAL n)

FOR i=0 TO 15
   PRINT Fibonacci (i);
NEXT i

FUNCTION Fibonacci(BYVAL n)
   SELECT CASE n
      CASE 0: Fibonacci=0
      CASE 1: Fibonacci=1
      CASE REMAINDER:
         Fibonacci= Fibonacci(n-1)+Fibonacci(n-2)
E     ND SELECT
END FUNCTION

FOR i=0 TO 15
   PRINT Fibonacci (i);
NEXT i
```

This is our final Fibonacci example (honestly!). A couple of points to note:

•The form of the DECLARE statement is exactly the same as the FUNCTION statement with the word DECLARE at the front.

•DECLARE statements are needed for two reasons. Firstly, they enable the compiler to check that the correct number and type of parameters have been used. Secondly if DECLARE was not used, the compiler might think that `Fibonacci(i)` in the example above was referring to an array `Fibonacci()`. In fact probably the only advantage of the FN syntax is that you can instantly see the difference between a function call and an array access. Of course, you could easily decide to use the similar conventions with FUNCTION definitions.

DECLARE statements can also be used for sub-programs. They are not required by HiSoft BASIC but can be useful for compatibility with other modern BASICs such as QuickBASIC. You can also use them as a documentation aid by having DECLARE statements at the front of your program for all the sub-programs and functions .

If you call a function which performs input/output inside another statement that performs input/output strange things may happen. There is no good reason for doing this and it should be avoided.



## Using the FN notation

In the past you could only define a function by using FN as a prefix to the function name; old-style BASICs restricted you even further in that the function could only be one line long. For example,

```
DEF FNrad(x)=x*3.141592653589793/180
```

which converts an angle in radians to degrees and could be used as follows

```
PRINT SIN(FNrad(45))
```

to give the sine of 45 degrees. The names of such user-defined functions must start with FN.

However, in HiSoft BASIC, DEF FN functions may have all the facilities of sub-programs with the following differences:

User defined functions return results by assigning to a pseudo-variable with the name of the function. For example

```
DEF FNfactorial(n)
   IF n<=1 THEN
      FNfactorial=1
   ELSE
      FNfactorial=n*FNfactorial(n-1)
   END IF
END DEF
```

which calculates the famous factorial function. Note that the definition finishes with END DEF and that on the right hand side of the assignment `FNfactorial` causes the function to be called again recursively.

The big difference between user-defined FN functions and sub-programs is that, in functions, parameters are call-by-value by default and to specify call-by-variable you should precede them with VARPTR. If you do not use variable checks, variables are assumed to be SHARED rather than STATIC. Naturally this difference can be confusing and so we recommend using the FUNCTION syntax instead.

Here is yet another coding of the Fibonacci example:

```
DEF FNfibonacci( n)
   SELECT CASE n
      CASE 0: FNfibonacci=0
      CASE 1: FNfibonacci=1
      CASE REMAINDER:
         FNfibonacci= FNfibonacci(n-1)+FNfibonacci(n-2)
   END SELECT
END DEF

FOR i=0 TO 15
   PRINT FNfibonacci (i);
NEXT i
```

Incidentally you can insert a space between the FN and the function name.

## Arrays and Sub-programs

Arrays may be used as parameters to sub-programs and user-defined functions. They are specified both in call statements and definitions with open and close parentheses after their names. The definition should contain the number of dimensions of the array. Arrays are always passed by reference.

```
DIM b(3,6)
MatSum b(),res

PRINT res

SUB MatSum(a(2),x)
STATIC i,j,x
   x=0
F  OR i=LBOUND(a,1) TO UBOUND(a,1)
      FOR j=LBOUND(a,2) TO UBOUND(a,2)
         x=x+a(i,j)
      NEXT j
   NEXT i

END SUB
```

This sums all the elements of the two dimensional array.



The corresponding function definition would be:

```
DIM b(3,6)
PRINT fnMatSum( b())

DEF fnMatSum(a(2))
STATIC i,j,x
   x=0
   FOR i=LBOUND(a,1) TO UBOUND(a,1)
      FOR j=LBOUND(a,2) TO UBOUND(a,2)
         x=x+a(i,j)
      NEXT j
   NEXT i
   fnMatSum=x
END SUB
```

Sub-programs may share arrays with the main program. The SHARED and DIM SHARED statements may be used as for scalar variables. The DIM SHARED statement when used with arrays also dimensions them. The SHARED statement should specify the number of dimensions of the array although this is not enforced.

For example,

```
DIM SHARED table(100)
'table() can now be access anywhere in the program.
```

or alternatively

```
DIM table(10)
table(10)=42 : Silly

SUB Silly
SHARED table(1)
   PRINT table(10)
END SUB
```

This will print 42.

## Local Arrays

Arrays may also be local to a sub-program and both STATIC and LOCAL varieties are supported. When using STATIC you need to make sure that the array is not dimensioned more than once. In the STATIC statement the number of dimensions may be included in parentheses.

For example,

```
'constants for Table Handler operations
CONST init=0, insert=1, find =2, replace=3

TableHandler init,0,0  'initialise the table
SUB TableHandler(operation, index, value)
STATIC table(1), first_free

   SELECT CASE operation
      CASE init
         DIM table(100)
         first_free=0

      CASE insert
      .
      .
```

In this example the array will only be dimensioned once, as long as the TableHandler sub-program is not called with a parameter of init more than once.

Using the LOCAL statement arrays may be created for the duration of this call to the sub-program. They are erased automatically at the end of the call. The actual dimensions are given in the LOCAL statement. For example:

```
SUB Recursive
'a temporary array with elements up to temp(40).
LOCAL temp(40)
.
.
END SUB
```

# Advanced Arrays

As well as the DIM, SHARED, STATIC and LOCAL statements described above there are a number of other array facilities that are not available in primitive BASICs.

The UBOUND and LBOUND functions return the size of arrays. See the example MatSum previously.



The lower-bound of arrays created by the DIM statement can be changed from the default value of 0 to 1 by the OPTION BASE statement, for example:

```
OPTION BASE 1 'arrays now start at one.
```

OPTION BASE is an executable statement, and so its effect depends on the order of execution in the program, not the order of the program text. It is thus possible for an array to have different dimensions if it is ERASEd and then REDIMmed. Using OPTION BASE normally only saves a considerable amount of memory if you are using 3 or more dimensions in an array.

When array checks are switched off OPTION BASE statements are ignored.

# Dynamic and staticarrays

Normally arrays that are DIMensioned using a constant value, for example DIM A(100) are treated as *static* arrays. Static arrays have a constant size specified by the DIM statement and may not be re-dimensioned or erased which means the compiler can generate code that will access them very quickly. If you have an array that you will subsequently want to re-dimension or erase then you'll need to use REM $dynamic which causes the compiler to make all DIMension statements declare dynamic arrays from this point in your program.

For example:

```
REM $dynamic
CONST arrsize=200
DIM arr(arrsize)
...
REDIM arrsize(300)
```

You can revert to the normal behaviour by using REM $static.

Dynamic arrays can grow and shrink in size and therefore HiSoft BASIC has to generate more code to handle this possible behaviour - if you want arrays to be accessed as quickly as possible using the minimum amount of code, use static arrays, which is the default. Static arrays are stored in the program's global space rather than on the heap.

Note that you must have used REM $DYNAMIC before you DIM the array for the first time; you cannot change the type of an array after it is declared.

## Other array features

ERASE works in two different ways, depending on the type of the array (dynamic or static). For static arrays, ERASE simply clears all the elements to 0 or to the null string. For dynamic arrays, ERASE may be used to free the space used by the array when it is no longer required. This is particularly useful if you have temporary results stored in an array. Once a dynamic array has been ERASEd you can DIM it again. For example,

```
REM $dynamic
DIM temp(10000) ' 10000 temporary results
.
.
ERASE temp

' note that temp is not followed by parentheses. This
' anomaly is present for compatibility with other BASICs
```

The REDIM statement for dynamic arrays gives the equivalent of an ERASE followed by a DIM in one statement. Thus

```
REDIM temp(100)
```

is equivalent to

```
ERASE temp: DIM temp(100)
```

REDIM cannot be used for static arrays.

HiSoft BASIC has a powerful extension to let you change the size of dynamic arrays whilst retaining their data called REDIM PRESERVE.

For example:

```
REM $dynamic

SUB AddElement(value)
SHARED table(1), maxentries, nextentry
   IF nextentry> maxentries THEN
   ' no room for this entry
      maxentries=maxentries+100
      REDIM PRESERVE table(maxentries)
   ' the above makes the array 100 elements larger
   END IF

   table(nextentry)=value  'enter the value
   nextentry=nextentry+1   'ready to store the next one

END SUB
```

This example shows how you can avoid fixed limits on the sizes of dynamic arrays. If you run out of room just make it bigger. REDIM PRESERVE requires enough memory to make a copy of the array. You can also use REDIM PRESERVE to make dynamic arrays smaller; again a copy of the array is made.

Normally the ERASE, REDIM and REDIM PRESERVE statements cause dynamic arrays to be moved in memory. As a result, if there are any pending array elements that have been used in variable parameters, then these will be become invalid. The best way to avoid this is by not passing array elements by reference. For example the following may not work as intended:

```
REM $dynamic
DIM x(50), a(30)
.
.
Subprog a(3) ' note variable parameter.
.
SUB Subprog(b)  ' note variable parameter

   ERASE x
      ' a() will now become corrupt, it has
      ' been moved because it was declared
      ' after x() which has been erased

.
.
END SUB
```

Unlike many BASIC compilers, HiSoft BASIC will let you change the number of dimensions of dynamic arrays with a REDIM statement. This may prove useful when porting certain programs that were developed with interpreters, however we recommend strongly that you avoid this as it can make programs almost unmaintainable.

For compatibility with earlier versions of HiSoft BASIC, you can use REDIM APPEND instead of REDIM PRESERVE. However we recommend the use of the latter as it is the form used by MicroSoft BASIC 7.1 on the PC.

# File Types

When you compile your programs to disk by clicking on the appropriate box in the Compile dialog box the compiler has to decide what file type to generate. If you have used any of the following commands and functions:

| | | | | | |
|---|---|---|---|---|---|
| INTIN | INTOUT | PTSIN | PTSOUT | CONTRL | GB |
| SYSTAB | VDISYS | GEMSYS | PSET | PRESET | POINT |
| WINDOW | CIRCLE | PCIRCLE | ELLIPSE | PELLIPSE | CLEARW |
| FULLW | LINEF | MOUSE | FILL | CLOSEW | OPENW |
| BAR | GET | PUT | | | |

then the program is made into a GEM program with extension .PRG.

The graphics versions of GET and PUT are the only ones that are affected; the file handling commands are not. If you use the GEMAES and GEMVDI libraries then .PRG will be assumed. If you have used the DA size option (J) then .ACC will be used; using DECLARE...CDECL forces generation of linkable code and thus a .O extension.

If none of the above statements and functions are used, then the type is assumed to be .TOS unless you have used COMMAND$ in which case it will be .TTP. The default extension of .TOS or .TTP can be over-ridden using the G (for GEM) compiler option. You can change the entire file name using the F option.



# Limitations Imposed by the Compiler

We have tried to avoid placing limits on the programs you can write. For example, most compilers have a limit on the number of characters that are significant in an identifier; HiSoft BASIC does not impose any limit on this so that

```
A_very_long_identifier_indeed_which_goes_on_and_on
```

is different from

```
A_very_long_identifier_indeed_which_goes_on_and_on_a
nd_is_different
```

This sort of limitation may not seem important to you, but such possible restrictions have the annoying habit of appearing when you think you have nearly finished a large program.

This section intends to list the remaining limitations other than the total workspace of the compiler. If you find these restrictive please tell us.

A program may not have more than 16383 lines. If you hit this limit you can probably get round it by having more than one statement per line. We have a 5000 line program which is about 135K bytes of source. If you exceed this you will be given a `Too many lines in program` compiler error.

The total number of active labels in the code generation phase of the compiler, as specified by the Max labels (H) may not exceed 5641. A label is generated for each line number or label that is referenced (not for those that are un-used) together with 2-3 or each sub-program, 2 for each CASE in SELECT statements plus 2 for the SELECT itself and 2 for most structure statements. For example our 5000 line program requires about 1100 such labels.

There is a limit of approximately 8000 on the number of sub-programs, shared and local variables and parameters in the entire program; if you exceed this you will be given a `Struture Table Full` error message.

The total number of different names in your program may not exceed 32767.

The total code of a SELECT statement may not exceed 32k bytes. To avoid this make some of the alternatives into sub-programs.

The total size of some FOR...NEXT loops may not exceed 32k bytes. To avoid this make some or all of the loop into a sub-program.

Sub-programs and user defined functions may not have more than 128 parameters.

The total space for global and STATIC local variables and the descriptor table may not exceed approximately 29K bytes. The amount of storage, in bytes, in this area required for the different types is:

| | |
|---|---|
| 2 | integers |
| 4 | long integers, single precision numbers |
| 8 | strings, double-precision numbers, all arrays |

The data in strings and arrays is not stored in this area. Static arrays are stored at the end of this static array but they are not subject to this 29K restriction.

Local variable stack space may not be more than 32k bytes per invocation. The different types require the number of bytes given in the table above.

There is a limit of 255 channels.

ON...GOTO and ON...GOSUB statements may not have more than 8190 line numbers each (!)



# Chapter 5
# The Libraries

## Introduction

A library is a collection of sub-programs and functions written in assembly language (with DevpacST/TT) which are available to your BASIC program just as if these routines had been written in BASIC in the first place.

The advantages of using libraries are:

- Simple BASIC programs that do not use libraries are kept small.
- You can use specific libraries for specific tasks thus improving the modularity and readability of your programs.
- Third parties can write and publish libraries for particular tasks e.g. MIDI.
- You can extend the power of HiSoft BASIC by writing your own libraries.

The task of this chapter is to explain how to use libraries and to detail which libraries have been supplied with HiSoft BASIC. We will present some small examples to help your understanding.

The *Technical Reference* manual details the contents of each library in *Chapter 2* and explains how to build your own library in *Appendix F.*

We have also supplied a source code library of high level GEM routines in the *HiSoft GEM Toolbox* which is described in *Chapter 3* of the *Technical Reference* manual. This is *not* a library as described here, but simply a collection of useful source code written in BASIC that allows you to program GEM more easily - in fact, the *HiSoft GEM Toolbox* makes extensive use of the operating system libraries and is a good example of how to get the best out of them.

If you want full control over the operating system you may well find that you need to use these operating system libraries rather than the *HiSoft GEM Toolbox*.

## The Libraries Supplied

As we have said, a HiSoft BASIC library is a file of routines to do a specific task. Every HiSoft BASIC library has an extension of .LIB. If you want to employ a particular library in your BASIC program you must use the `LIBRARY` statement to name the library that you wish to use e.g

```
LIBRARY "gemvdi"
```

The libraries that we have supplied with HiSoft BASIC are mostly connected with handling the ST/TT operating system:

| | |
|---|---|
| BIOS.LIB | The Basic Input/Output System |
| GEMAES.LIB | GEM Application Environment Services |
| GEMDOS.LIB | The operating system |
| GEMVDI.LIB | GEM Virtual Device Interface |
| XBIOS.LIB | The eXtended Basic Input/Output System |

The operating system is made up of GEMDOS which uses the BIOS and XBIOS and GEM which consists of the AES and the VDI. We will give an overview of these libraries followed by some short examples of their use.

One important fact to remember when using these libraries is that you should not use library function names as names of your own variables within your program.

## Operating System Overview

The operating system is the program that sits between the user's program and the hardware of the computer. Its purpose is to provide standard routines for using the features of the computer. In the case of the ST/TT range of computers, the operating system features range from the general, e.g. printing a character on the screen, to the extremely hardware specific, e.g. setting a bit in a specific port of the sound chip.



This is an advantage for the programmer as he/she does not have to write many of the routines to access the hardware directly; the capability is provided by the operating system.

The three parts of the operating system are organised in a vaguely hierarchic manner. At the highest level is GEMDOS. It provides control over file handling, I/O with devices, memory allocation, and program execution. The BIOS is frequently called by GEMDOS to perform tasks. The prime examples of BIOS routines are reading data from storage devices, and lower level console and interface I/O. The XBIOS is the explicitly hardware specific part of the operating system. It handles I/O to the floppy disk drives and various chips of the computer, and provides support functions such as the generation of boot sectors in memory.

## Examples

Here are some examples of using the operating system libraries.

## DiskCopy Example Program

The `DISKCOPY` program in the `EXAMPLES` folder on your master disk uses calls to the BIOS and the XBIOS directly; it is a prime example of operating system library use.

In the beginning of the source we have the line

```
library "BIOS","XBIOS"
```

This declares that routines in these two libraries are used in the program.

The first usage of a library routine is the BIOS `bconstat%(2)` call. This checks if a character is available from the keyboard. When a character becomes available, `bconin&(2)` is used to get the character from the keyboard; otherwise it would be buffered.

`floprd%`, `flopfmt%` and `flopwr%` are some of the XBIOS calls to deal with the disk drives. In the case of `DISKCOPY`, they are used within sub-programs to promote readability of the actual copying algorithm. The sub-programs are simple enough: they provide screen output to keep the user informed of what is being done to which track and one line calls the XBIOS with the relevant parameters.

Given the way that the program is designed, it is easy for you to add capabilities such as single or double sided disks, user selected source and target drives and single drive copying. This is left as an exercise for the reader.

## Dir2String Example Program

This program will copy the names of files in a given directory into a string. To do this, the GEMDOS library is used. The actual routine looks like this:

```
sub dir2str(pathstr$,targstr$)
static isitthere,olddta&,addr&,hold
  olddta&=fgetdta&
```

GEMDOS returns the old Disk Transfer Address. The DTA points to a buffer where GEMDOS keeps information when searching for files.

```
  fsetdta varptr(dta(0))
```

We want to use our own buffer to avoid any possible conflict with the operating system. `dta()` was previously dimensioned in the main program.

```
  isitthere=fsfirst%(pathstr$,0)
  if isitthere<0 then
     fsetdta olddta&
     exit sub
  end if
```

This is the search for the first occurrence of a file that matches what is contained in `pathstr$`. If `isitthere` is negative, no file was found, the old DTA is restored and control is returned to the routine that called the sub-program.

If a file was found the following code comes into action.



```
     do
        addr&=varptr(dta(15))
        hold=peekb(addr&)
        if hold=0 then exit loop
        targstr$=targstr$+chr$(hold)
        incr addr
     loop
```

The null-terminated filename is extracted from the buffer and put into `targstr$`.

```
     targstr$=targstr$+chr$(13)+chr$(10)
```

A CR-LF is inserted after every file name so things are more legible.

```
     isitthere=fsnext%
     if isitthere<0 then
        fsetdta olddta&
        exit loop
     end if
  loop
end sub
```

Here again is the check if another file can be found; if not, the old DTA is restored and the sub-program is over.

As you can see, using the libraries is not hard at all. The major pitfall is understanding exactly what each of the calls does. Experimenting is not a good idea in the case of low level disk accesses, as you can easily crash your system, delete data on media, or do other potentially destructive things. For more detailed documentation of the operating system and its functions please consult the *Bibliography*.

## Loading and Using Resource Files

Unlike most languages for the ST/TT, HiSoft BASIC will let you create menus without using a resource editor. However the easiest way to create dialog boxes or your own icons is using our resource construction set, WERCS, together with the *HiSoft GEM Toolbox*.

The following is an example of how to load a resource file that we supply for you (in the EXAMPLES folder) and use it from within HiSoft BASIC; this example uses only the operating system libraries and, as such, it is a good contrast to using the *HiSoft GEM Toolbox* - see the *Tutorials* chapter for a simpler way of loading a resource file.

```
DEFINT A-Z
LIBRARY "GEMAES"
REM $OPTION L8

IF rsrc_load("BASIC.RSC")=0 THEN
   PRINT "Can't load resource file": STOP
END IF
```

The `REM $OPTION L8` command is used to "leave" 8k of RAM for the operating system so that it has plenty of room to load the resource file.

In this resource file the object tree with index 2 is the Display BASIC line number dialog box. To find out where this has been loaded we use the `rsrc_gaddr` routine. This can then be centred on the screen using `form_center`. When preparing to display a dialog box it is a good idea to call the `form_dial` sub-program with a parameter of 0. Then you draw the object tree using `objc_draw` and let the user type and click with `form_do`. Finally call `form_dial` with a parameter of 3; this makes the operating system send messages to update windows.

So a complete program to display and let the user interact with this dialog box is:

```
DEFINT A-Z
LIBRARY "GEMAES"
REM $OPTION L8

IF rsrc_load("BASIC.RSC")=0 THEN
   PRINT "Can't load resource file": STOP
END IF

junk=rsrc_gaddr(0,2,form&)    ' 0 for object tree
          ' 2 item number
' form& now contains the address of the object in memory.

form_center form&,x,y,w,h
' x,y,w,h gives centred position

form_dial 0,0,0,0,0,x,y,w,h
a=objc_draw(form&,0,3,x,y,w,h)
a=form_do(form&,2)
form_dial 3,0,0,0,0,x,y,w,h

PRINT a
```

This will print out the number corresponding to the final button clicked on. The number that is entered in the dialog box is stored within the object tree.

Note the use of 3 in the call to `objc_draw` to draw 3 levels of the tree structure so that it is all printed. The 2 in the `form_do` command is number of the text object where the cursor appears initially.

How do we know that this dialog box is object number 2 in the tree? It was created using a resource editor, which had the ability to write a file with constant definitions in it to give the numbers of the items.

## The C Header File Converter

HiSoft WERCS will produce .BH files for use with HiSoft BASIC directly. However if you have another resource editor which produces header files of constants in the C language, we provide a tool to convert C header files to HiSoft BASIC header files. The only C constructs supported are `#define` and comments.

To run the converter program double click on CTOBAS.TTP and type the name of the C file. You can leave out the .H if you like. This will produce a file in the same directory as the source file but with an extension of .BH (for BASIC Header).

## The AES Constant File

To make your AES programs more readable we suggest that you use the constants in the GEMAES.BH; these include most of the constants in the Digital Research documentation and the byte offsets for the data structures used for object trees.

# Appendix A
# Converting Programs

## Introduction

This Appendix is intended for use by programmers wishing to convert various forms of BASIC into HiSoft BASIC. It starts with general notes about all conversions, then is sub-divided into sections for particular common BASIC implementations. It finishes with some hints and tips on writing programs that can easily be ported to the IBM PC, Macintosh and Amiga

## General Conversions

Most BASICs share a certain core of the language, no matter how different they seem to be. `PRINT` always does the same thing, so does `INPUT` and so forth. BASICs can differ by being machine specific, ANSI standard, or conforming to a certain style (MicroSoft's QuickBASIC and Borland's Turbo Basic are notable examples).

Whichever BASIC you are trying to convert a program from, you will need to get the source into ASCII. Some BASICs (e.g. ST BASIC) use ASCII for their programs anyway, but most interpreters use a tokenised form that will come out as gibberish if loaded into the editor. You should save your program from your BASIC interpreter using an ASCII option, if possible.

Some things stand very little chance of being the same from one BASIC to another, in particular internal floating point representations and random-access file formats.

One thing to be aware of particularly is that string length limitations found in other BASICs do not exist in HiSoft BASIC. In particular the `LEN` function can return a long integer as a result, not just an integer as in other BASICs.

# HiSoft BASIC 1

The following is a summary of some potential incompatibilities between HiSoft BASIC 1 and version 2; don't panic at the size of this list as the majority of HiSoft BASIC, Power BASIC and FirST BASIC programs will compile without change. We have listed these under headings of facilities used.

## LIBRARY statement

Ensure that you have not selected the `No "FN"s in library` option otherwise your calls to library functions will not be recognised.

## GEMAES library MENU& function

This function has now been moved to its own library appropriately called MENU. So add

```
LIBRARY "menu"
```

near the top of your program.

## REDIM/ERASE of arrays

If you dimension an array using just a number or a constant (e.g. `DIM A(1000)` you can't `ERASE` or `REDIM` it in HiSoft BASIC 2 unless you add

```
REM $DYNAMIC
```

before the `DIM` statement. If the `DIM` expression is a non-constant expression e.g.

```
array_size=100
DIM A(array_size)
```

then there is no need for this change. As well as enabling the compiler to produce faster code for static arrays, this change also brings HiSoft BASIC in line with the MicroSoft QuickBASIC usage.



## OPEN...FOR OUTPUT

Under earlier versions of HiSoft BASIC for the Atari, sequential files were automatically terminated with a Ctrl-Z character (`CHR$(26)`). This is no longer the case by default, but can be enabled using

```
POKEB SYSTAB+67,1
```

## ERL function

If no logical line numbers have been used, this function will now return the physical line number that caused the error. If your `ON...ERROR` routine is testing to see if `ERL` is 0 (perhaps to see if you are still in your initialisation code) then you'll need to insert a line number at the start of your program and test for this value instead.

This feature was introduced so that modern programs that need to use an `ON...ERROR` to ensure safe termination (freeing operating system resources for example) can give an indication of the source of the error. This facility is however not available in MicroSoft BASIC on the PC.

## MKD$,CVD functions

These functions have been changed so that they now return values that are compatible with HiSoft BASIC on the Amiga and our other languages on the Atari. This change will generally effect programs that store doubles in random access files. If you want to use data files that were created with an older version of HiSoft BASIC for the ST immediately then you can use the `Use old style doubles in files` (#) option but in general we recommend that you convert your data files to the new format.

## REM $OPTION R

The return stack size option now takes a value in *kilobytes* so that you can write programs that use recursion intensively. If you are already using this option you'll need to change it as older versions specified this as a number of bytes. If the value used is greater than 200 (the old minimum value) the compiler will give you a warning message.

## REM $OPTION D

This option now selects extended rather than DRI debugging information. You'll want to use option S as well to output you labels to disk. Note that sub-progam and function names are now given the identifiers that you used in your program. rather than names created by the compiler.

## GEMVDI library: FNhandle

The GEMVDI library function `FNhandle` has been renamed `cvhandle` so that you can use the variable handle in conjunction with the `No "FN"s in libraries` option.

## New reserved words

The following are now reserved words: `ALIAS`, `BIN$*`, `BYVAL`, `CDECL`, `CURDIR$`, `DECLARE*`, `ENVIRON`, `ENVIRON$`, `FORMATD$`, `FORMATI$`, `FORMATL$`, `FORMATS$`, `FREEFILE`, `FUNCTION*`, `GETCOOKIE`. `IS`, `LTRIM$`, `MIN`, `MAX`, `PRESERVE`, `RINSTR`, `RTRIM$`, `SPEEKB`, `SPEEKL`, `SPEEKW`, `SPOKEB`, `SPOKEL`, `SPOKEW` and so can no longer be used as names for user variables, sub-programs and labels. Those marked with an asterisk (*) in the above list were added in HiSoft BASIC 1.3.

If you are in a hurry to compile your existing HiSoft BASIC 1 program then you can include HBST10.BH or HBST13.BH at the start of your program and then these reserved words will be disabled. Use HBST13.BH if your old program was for HiSoft BASIC or Power BASIC 1.3 or above; otherwise use HBST10.BH.



# Atari ST BASIC

In the dim, distant past ST BASIC was supplied free with the Atari ST computers and therefore we have endeavoured to be as compatible as we can with it, even to the extent of emulating one of the bugs! However we naturally have included many extensions to the language.

One major difference is that double-precision numbers are fully supported in HiSoft BASIC, unlike ST BASIC which pretends it has doubles but in fact only has singles internally. The limits of ST BASIC, such as strings up to 255 characters, arrays only taking a third of available memory and limited to 32k, and so, on are not relevant once the program has been compiled. The interpreter produces .BAS files in ASCII which are directly loadable by the HiSoft BASIC editor.

The bug we have deliberately emulated concerns the `GOTOXY` statement as so many existing programs rely on it. We have however *not* emulated the bugs in `INT` which, under ST BASIC, returns the same result as `FIX` and `INKEY$` which always returns a null string. The `FLOAT` function in ST BASIC has absolutely no effect so we do not compile it. The `CALL` statement equivalent in HiSoft BASIC is `CALL LOC` and the `RESET` command acts rather differently.

In addition to the statements listed in the *Command Reference* section HiSoft BASIC also supports the following functions and statements which produce the same results as ST BASIC:

`CONTRL`, `CLEARW`, `CLOSEW`, `FULLW`, `GB`, `GEMSYS`, `INTIN`, `INTOUT`, `OPENW`, `PTSIN`, `PTSOUT`, and `VDISYS`.

For documentation on these see your ST BASIC manual. We recommend strongly the use of the `WINDOW` statement and the VDI and AES libraries in preference to the ST BASIC statements.

HiSoft BASIC does not support the statements in the interpreter that have no meaning in a compiler environment:

`AUTO`, `BREAK`, `CONT`, `DELETE`, `DIR`, `EDIT`, `ERA`, `FOLLOW`, `LIST`, `LLIST`, `LOAD`, `MERGE`, `NEW`, `OLD`, `QUIT`, `RENUM`, `REPLACE`, `SAVE`, `TRACE`, `UNBREAK`, `UNFOLLOW` and `UNTRACE`.

HiSoft BASIC does not support the following ST BASIC statements:

COMMON, RESUME NEXT and WAIT.

## Old-Style Microsoft BASICs

Microsoft BASIC is the closest to a world standard for the BASIC language and is the one around which we designed HiSoft BASIC.

For this reason most versions of Microsoft BASIC should not present problems converting to HiSoft BASIC. Programs written in the old-style BASICs, such as those found in Commodore 64s and under CP/M should require little or no work to convert, as long as machine-specific PEEKs and POKEs are avoided. You can also save typing by not entering the line numbers that aren't needed.

Most BASIC interpreters from other vendors are at least in part based on the same principles as Microsoft so should also convert reasonably easily.

## New-Style Microsoft BASICs

The new style of Microsoft BASIC is defined by QuickBASIC, Visual BASIC and the MicroSoft BASIC PDS and running on IBMs and compatibles. By *new-style* we mean support for structured programming such as sub-programs and parameter passing, CASE, , DO etc. HiSoft BASIC supports almost all the features of QuickBASIC 2 and 3 and most of the features of QuickBASIC 4.0, 4.5 and has some additional features from the PDS 7.1 system.

The main advantages of HiSoft BASIC that you can exploit when converting programs from the IBM world, are the large memory and greater graphic support, in particular GEM. In addition, recursive programming techniques can be used.

When converting QuickBASIC programs there are a few things which are not supported by HiSoft BASIC by reason of operating system or hardware differences.



Missing statements under this category are:

COM, CVSMBF,CVSMBF,DRAW, ERDEV, ERDEV$, FILEATTR IOCTL$, IOCTL, KEY, LINE, LOCK, MKDMBF$,MKSMBF$, ON various, PAINT, PEN, PLAY, PMAP, SETMEM, SHELL, UEVENT, UNLOCK, VARPTR$, VARSEG, VIEW, and WAIT

Obviously PEEKs and POKEs are completely different, as are machine-code calling conventions. There are also slight differences in the following commands:

CIRCLE, CLEAR, CLS, COLOR, FILES, LOCATE, OPEN, PCOPY, POINT, SCREEN, SEEK, SOUND, STICK, and WINDOW.

HiSoft BASIC does not at this time support the following Microsoft BASIC statements:

COMMON , DOUBLE, INTEGER, LONG, RESUME NEXT, SINGLE, TYPE...END TYPE,

In addition to this high degree of Microsoft compatibility, HiSoft BASIC also compiles many of the additional features found in Borland's Turbo Basic compiler and Spectra Publishing's Power BASIC for the PC.

## HiSoft BASIC on the Amiga

The main area of incompatibility here is in graphics commands. The Atari version does not support the following AmigaBASIC commands:

AREA, AREAFILL, BREAK. COLLISION, CVFFP, LIBRARY, MENU, MKFFP$, OBJECT . various, ON... various, PAINT,PTAB, SAY, SCROLL, TRANSLATE$

The following commands are implemented on both machines but differ in their action:

BAR , CIRCLE, COLOR, FRE, PALETTE, SCREEN, SOUND, SYSTAB, WAVE, WINDOW.

The following are reserved words in HiSoft BASIC 2 but may be used as variables on the Amiga; either change the variable names or use the HBAM2.BH include file - this will prevent you from accessing the following ST BASIC commands:

CLEARW, CLOSEW, ELLIPSE, FILL, FULLW, GB, GEMSYS, GETCOOKIE, LINEF, OPENW, PCIRCLE, VDISYS

In addition, we also supply HBAM1.BH which excludes the reserved words that were introduced in version 2 of HiSoft BASIC for the Amiga.

# Fast BASIC

Fast BASIC achieved a popular following on the ST, and deservedly so. It is easy to use, allows good control over GEM and is reasonably fast by interpreter standards. Unfortunately it was not an attempt to follow the Microsoft path but instead BBC BASIC was chosen as a model. As a result there are a few non-trivial differences to be wary of when converting Fast BASIC programs to HiSoft BASIC.

Many AES and VDI calls have specific statements set up in Fast BASIC. These do not necessarily have the standard routine names or calling conventions (parameters passed etc.). Any program that makes use of Fast BASIC's GEM features must be changed to use the HiSoft BASIC AES and VDI libraries. The same problem exists for GEMDOS, BIOS and XBIOS calls. Many Fast BASIC commands simply call the operating system. Usage of these must also be converted to use of the supplied operating system libraries and a good idea of which routine to use can be found in the Fast BASIC manual after each command description.

Fast BASIC file handling is extremely ST specific. It uses the file handles returned by the operating system; these handle numbers can, for instance, be stored in variables. This form of file handling is not in the least compatible with the Microsoft style file handling system which uses channels.



As far as general syntax is concerned, Fast BASIC doesn't deviate terribly much. The control structures are slightly different and variable type specifiers use different characters. As Fast BASIC has so many reserved words it is case sensitive with respect to variable names, whereas HiSoft BASIC is not, which may require some alteration.

The program FASTCONV is supplied in source form in the EXAMPLES folder; it is a Fast BASIC to HiSoft BASIC converter. The list of words that are replaced appears early on in the program; various special cases (e.g. DEFs and various character substitutions) are dealt with later.

There are few features of the converter which are important to know about:

FASTCONV works only on ASCII files. Normal Fast BASIC sources are held in a tokenised format completely unintelligible to FASTCONV (or anything else apart from Fast BASIC itself).

To produce an ASCII file you have to drag the icon of the source file on the Fast BASIC Desktop to the Clipboard. Then drag the Clipboard icon to the Disk Drive icon. A file selector will appear and you should enter the name of your file (the default extension for ASCII Fast BASIC files is .ASC). Be careful not to over-write your original source accidentally (i.e. the .BSC file). The .ASC file produced can be processed by FASTCONV.

It is important to remember that FASTCONV cannot convert every Fast BASIC program completely. It is intended to relieve the programmer of the tedious replacement of keywords and variable type specifiers. The source of FASTCONV has been designed so that it is relatively easy to add features to the program; it also serves as an example of structured and modular BASIC programming.

# ST BASIC Version 2

ST BASIC 2 is approximately compatible with version 1, though the additional features of ST BASIC 2 are not directly supported by the HiSoft BASIC compiler.

The main difference as far as compiling ST BASIC 2 programs is that %s used after variable identifiers should be changed to &s, as % in ST BASIC 2 means long-integer. There now follow notes for converting the ST BASIC 2 specific features into HiSoft BASIC.

### AREA

The points should be put in an integer array then the VDI library routine `v_fillarea` used.

### ASK MOUSE x,y,b

should be changed to

```
x=MOUSE(0): y=MOUSE(1): b=MOUSE(2)
```

### ASK RGB reg,r,g,b

should be changed to use the VDI library:

```
vq_color reg,0,rgb%(): r=rgb%(0): g=rgb%(1): b=rgb%(2)
```

### BIOS

should be converted into a suitable BIOS library call.

### BOX x1,y1;x2,y2

should be converted into `BAR x1,y1,x2-x1,y2-y1`

### DRAW

should be converted to use the VDI library routine `v_pline`

### DRAWMODE x

should be converted to use the VDI library routine `vswr_mode x`

### ERR$

has no equivalent.

### GEMDOS

should be converted into a suitable GEMDOS library call.



### GSHAPE x,y,array()

should be converted to `PUT x,y,array`

### LINEPAT x

should be converted to use the VDI library routine `vsl_type x`

### LINEPAT 7,p

should be converted to `vsl_type 7: vsl_udsty p`

### MAT AREA c,array%()

should be changed to use the VDI routine `v_fillarea c,array%()`

### MAT DRAW c,array%()

should be changed to use the VDI routine `v_pline c,array%()`

### MAT LINEF see MAT DRAW

### MAT SOUND array%()

should be converted to use the XBIOS library routine

```
junk&=FNdosound(VARPTR(array&(0))
```

Note that the array in ST BASIC 2 is denoted with % and *must* be converted to & in HiSoft BASIC.

### PATTERN p,a%()

should be converted to use the VDI library routine `vsf_updat a%(),p`

### RGB reg,r,g,b

should be converted to use the VDI library routine `vs_color reg,r,g,b`

### SSHAPE x1,y1;x2,y2,array%()

should be converted into `GET (x1,y1)-(x2,y2),array%()`

though you should ensure the array you are using is of the correct size for HiSoft BASIC.

### STATUS

is normally equivalent to the return values of the function calls in HiSoft BASIC library routines.

### XBIOS

should be converted to a suitable XBIOS library routine.

### a=PEEK_B(x)

If `x` is not protected memory then use `a=PEEKB(x)` else use `a=SPEEKB(x)`.

### a=PEEK_W(x)

If `x` is not protected memory then use `a=PEEKW(x)` else use`a=SPEEKWx)`.

### a=PEEK_L(x)

If `x` is not protected memory then use `a=PEEKL(x)` else use `a=SPEEKB(x)`.

### POKE_B, POKE_W, POKE_L

Similar to `PEEK` above except use `POKE` equivalents.

### GEMSYS

Replace with `GEMSYS(x)` where `x` is the number placed into `GEM_CONTRL(0)` in the ST BASIC 2 program.

### VDISYS

Replace with `VDISYS(0)`.



### CONTRL, PTSIN, PTSOUT, INTIN, INTOUT

In ST BASIC 2 these are arrays and should be converted into `PEEKW` or `POKEW` statements. For example the ST BASIC 2 statements

```
CONTRL(0)=11: CONTRL(1)=42: PTSIN(2)=3: x=PTSOUT(3)
```

should be converted into

```
POKEW CONTRL,11: POKEW CONTRL+2,42: POKEW PTSIN+4,3
x=PEEKW(PTSOUT+6)
```

Note that the array index *must* be doubled before adding.

### GEM_ADDRIN, GEM_ADDROUT, GEM_CONTRL, GEM_GLOBAL, GEM_INTIN, GEM_INTOUT

In ST BASIC 2 these are arrays and need to be converted into `PEEKW`s and `POKEW`s using additional long-integer variables set up like this:

```
GEM_CONTRL&=PEEKL(GB)
GEM_GLOBAL&=PEEKL(GB+4)
GEM_INTIN&=PEEKL(GB+8)
GEM_INTOUT&=PEEKL(GB+12)
GEM_ADDRIN&=PEEKL(GB+16)
GEM_ADDROUT&=PEEKL(GB+20)
```

These can then be used in `PEEKW` & `POKEW` statements in a similar way to that described previously.

## Writing for compatibility

We are often asked which compiler you should purchase to make it easy to port a program that you've written in HiSoft BASIC on the Atari if you want to move it to another system. At the time of writing we recommend the following:

For the PC, MicroSoft's BASIC PDS 7.1 or the much cheaper, but less sophisticated, QuickBASIC 4.5.

For the Amiga, HiSoft BASIC of course.

For the Macintosh, MicroSoft QuickBASIC. At the time of writing this is the least advanced of the above.

If you want to write your program so that it is easy to port to other systems, here are a few hints:

- Isolate your use of system specific features to a few sub-programs. If you can, try porting these to other machines before you use them extensively; you may find that you are relying on something that is easy on the Atari, but different on another machine.
- If you are porting to a non-HiSoft product be wary of limitations on the sizes of strings and arrays. QuickBASIC 4.5 doesn't allow more than 64K of strings total for example and many BASICs won't let you have more than 32767 elements in an array.
- Use the forms of statement that we recommend in the *Command Reference* section. For example, use `EXIT DO` rather than `EXIT LOOP` and `REDIM PRESERVE` rather than `REDIM PRESERVE`.



# Appendix B Hints and Tips

This chapter shows how you can get the most out of programs written with the HiSoft BASIC compiler. It is not necessarily intended only for the advanced programmer.

## Using HiSoft BASIC

Making a more efficient program requires knowledge of the features of HiSoft BASIC. The following suggestions are intended to give you a firmer understanding of what can be done; we also hope that you will use this information as the basis for a more detailed exploration of HiSoft BASIC.

### defint a-z

It is a good idea to have this line in your program, it makes the default variable type a short integer. The main benefit from using `int`s is speed. The source becomes more understandable when `&`, `!`, and `#` are used explicitly.

## rem $option v+

This forces variable checks on. The primary benefit of this is that you can avoid unnecessary bugs caused by undefined variables in sub-programs and functions.

## STATIC variables in SUBs and FNs

`STATIC` variables retain their values when a `SUB` or `FUNCTION` is exited and re-entered. There is a speed benefit in using `STATIC`s as opposed to `LOCAL`s: `STATIC`s are only allocated once, whereas `LOCAL`s must be allocated every time a `SUB` or `FUNCTION` is invoked. If your `SUB` or `FUNCTION` is recursive then `LOCAL` variables *must* be used.

## INCR and DECR

These two statements respectively increment and decrement the value of a variable by one. When used on array elements they are considerably faster than e.g. `Arr(1)=Arr(1)+1`.

## ==

The double-equals comparison operator has a different meaning depending on the type of value compared. When comparing strings, a case-independent comparison is made. It is considerably faster than using UCASE$ or LCASE$ and then comparing, or doing something like `IF fred$="A" or fred$="a" THEN...` When comparing numeric values, == is used as 'almost equals'; rounding errors can thus be avoided. There is a performance degradation when using == on numeric values. Note that this is a HiSoft extension.

## ! as opposed to #

When using floating point maths, it is a good idea to know exactly how much accuracy is actually necessary. Single-precision is much faster than double-precision, due to the degree of accuracy required. If you want floating point, but do not need such a high degree of precision, single-precision is the variable type to use.

## VARPTR, SADD and PEEKtype

When using VARPTR and SADD, you must be very careful. The reason for this is quite simple: owing to the dynamic heap allocation and the blindingly-fast garbage collection of HiSoft BASIC, strings are prone to move around without any prior notice. This is normally completely transparent to the user, but when using these functions it becomes a factor to be reckoned with. If you are going to use them, call the functions just before accessing the variable or array. In addition the address of any array will be invalid if a REDIM or ERASE statement has been executed since VARPTR was used to find the address.



After using these functions, a PEEK or POKE is usually inflicted upon the data at the returned address. HiSoft BASIC has faster varieties of PEEK and POKE and they are type-specific. The can be used to great effect e.g. when parsing a string.

Let us suppose an entire file has been loaded into a single string using `INPUT$`; this is quite possible because strings in HiSoft BASIC are only limited by available memory. SADD is called to determine where the string is. To go through the file at high speed, all you need is to remember your place in the string and PEEKB from the location that you need. Do not use PEEKW or PEEKL because strings can be on odd boundaries and an address exception will occur if you try to read a word or long from an odd address.

## Profiling, SUBs and FUNCTIONs, and speed

Profiling is a powerful feature whereby you can find out where a program is spending its time. This is extremely useful when the program is finished and you want to upgrade the performance of the program as much as possible. For exact details about HiSoft BASIC's profiling features, please see *Appendix H* in the *Technical Reference* manual.

Using PROFILE.TTP you can find out which lines are executed most frequently. If you use sub-programs and functions, you can see which of these takes up the most time as well. Not only for this reason is it a good idea to use sub-programs, functions, SELECT CASE statements, DO...LOOPs and the many other structure elements of HiSoft BASIC. The more you use these elements of the language, the more readable and modular your programs become. Sub-programs that you develop for one program can conceivably be used in another program by simply inserting the SUB.

If you have found a routine that is extremely time-critical, but cannot be sped up within the context of BASIC, another alternative is to re-write the routine in assembly language or C. This is quite easy if you know 68000 assembler. For details on how to do this, please consult *Appendix F* of the *Technical Reference* manual.

If you have decided that you wish to write a routine in assembly language, make sure that you have a working algorithm in BASIC. We have found that it helps immensely if the algorithm is worked out and debugged in BASIC, and then transcribed into assembler. Using this method, bugs can be avoided that would take longer to find in assembler than in BASIC.

## Programs that use graphics

Since the ST has three available screen resolutions, the TT an additional 3 resolutions and there are a number of third party graphics add-ons available, the quality of life of a graphics programmer is both enhanced and complicated. On the positive side, the most suitable resolution can be chosen. On the negative side, programs really ought to be able to run in as many resolutions as possible; this makes more work for the programmer.

In general, it is best to find the sizes of windows etc. (e.g. `WINDOW GET`), rather than try to find out the resolution in which you are working. You'll be amazed that a general version that can cope with any resolution will often be shorter and easier to understand than one that is purely designed for ST resolutions.

If your program needs some hardware specific feature to function, first check that this is available using the `GETCOOKIE` function.

## Making Your Programs "No-Limits"

If you have been used to programming in more primitive versions of BASIC or in Pascal or C, avoiding arbitrary restrictions on the size and type of data that your program can manipulate can be hard work. For example, having a limit on the length of names that you can type into a business application can sometimes be very annoying. Similarly avoiding limits on the lengths of files and in line lengths can save you a lot of time, if say the file in question has odd end-of-line markers that mean that your program treats the whole file as one line.



The following hints should help to avoid this sort of problem:

When reading in or adding to arrays, have code to make the array larger if need be. In general adding a few elements at a time is not a good idea because the program may start spending all its time moving arrays. Normally it is a good idea to start off an array with the size as just larger than a typical requirement, but if an array is only used occasionally then the dimension can start off small. For example, if writing a cross-reference program for HiSoft BASIC programs, it would probably a good idea to start by assuming that the number of line numbers is small (say 10) and then if the program turns out to be a horrible old-fashioned program with a line number on every line then the arrays used for holding them can be grown using `REDIM PRESERVE` at, say, 100 elements at a time.

When using byte or record numbers in files or strings use long integer variables (terminated with `&`). This should remove automatically many 32k or 64k limits on programs that are designed with 16-bit integers in mind.

# Appendix C
# Technical Support

HiSoft BASIC comes with 30 days free technical support, starting from the date of registration; therefore you should send in your registration card quickly. Technical support is available by telephone during our Technical Support Hour, by letter or by fax.

Should you wish to receive extended technical support, please complete the relevant sections on the registration card, indicating whether you would like to take up the *Silver* or the *Gold* service.

In addition to your name, address and postcode (very important for UK customers), we need payment details before we can accept your extended registration. You can pay by credit card (Mastercard, Eurocard, Access, Visa etc.), UK debit card (Switch, Connect etc.), Eurocheque, UK cheque or Postal Order.

You may have already registered another HiSoft product under our *Gold* or *Silver* service; in this case, there is no need to fill out the payment section.

# Appendix D Bibliography

This bibliography contains our suggestions for further reading on the subject of the ST/TT, BASIC, and GEM. The views expressed are our own and as with all reference books there is no substitute for looking at the books in a good bookshop before making a decision.

## ST/TT

Unfortunately, there are now a limited number of good books on the ST/TT and GEM - this is a selection that we know.

*A Hitchhikers Guide to the BIOS*

*Atari Corp [1986]*

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

This is the standard developer's document which is supplied with the official developer's package. It documents the BIOS and XBIOS, as well as correcting errors in the DR GEMDOS document. It is well written and updated when necessary.

*Atari GEMDOS Reference Manual*

*Atari Corp [1986]*

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

*Atari ST Internals 3rd Edition*

*Brückmann, Rolf, Lothar Englisch and Klaus Gerits [1988]*

ISBN 0-916439-46-1, Data Becker GmbH, Merowingerstraße 30, 4000 Düsseldorf, West Germany.

This book is the best documentation available for the user who is not a registered ST developer. It is not suitable for the beginner, as it describes the hardware and operating system of the ST. The major fault with this book is that the BIOS listing is out of date and it contains some considerable inaccuracies.

### *COMPUTE!'s Technical Reference Guide, Atari ST - Volume I: VDI*

*Sheldon Leeman [1987]*

ISBN 0-87455-093-9, COMPUTE! Publications, Inc., P.O. Box 5406 Greensboro, NC 27403, USA.

### *Concise Atari ST 68000 Programmer's Reference*

*Katherine D. Peel [1986]*

ISBN 1-85181-017-X, Glentop Publishers Ltd., Standfast House, Bath Place, High Street Barnet, Herts EN5 5XE, U.K.

An alternative to Atari ST Internals. It contains information on the ST's hardware, the operating system and GEM. Its coverage of the various levels of the machine is comprehensive, though a couple of sections are very inaccurate and some features are described that simple don't exist.

It is rather difficult to find one's way around as the layout is based on large numbers of tables and it lacks an index.

### *GEM Programmer's Guide, Volume 1: VDI*

*Digital Research [1987]*

Digital Research Inc., 60 Garden Court, P.O. Box DRI, Monterey, CA 93942, USA.

### *GEM Programmer's Guide, Volume 2: AES*

*Digital Research [1987]*

Digital Research Inc., 60 Garden Court, P.O. Box DRI, Monterey, CA 93942, USA.

### *GEMDOS Extended Argument (ARGV) Specification*

*Atari Corp [1989]*

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.



### *Professional GEM*

*Oren, Tim [1985]*

ANTIC Publishing

### *Programmers Guide to GEM*

*Balma, Phillip and William Fitler [1986]*

ISBN 0-89588-297-3, SYBEX Inc., 2344 Sixth Street, Berkeley, CA 94710, USA.

### *Rainbow TOS Release Notes*

*Atari Corp [1989]*

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

### *S.A.L.A.D. - Still Another Line A Document*

*Atari Corp [1987]*

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

### *STE TOS Release Notes*

*Atari Corp [1989]*

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

### *The Pexec Cookbook*

*Atari Corp [1989]*

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

## GEM

Unfortunately, there are now a limited number of good books on GEM - this is probably the best available.

### *GEM on the Atari ST*

*Published by Abacus/Data Becker*

Describes programming under GEM. It is not as complete as the DR manual and contains similar errors.

### *GEM Programmer's Guide Volumes 1 & 2 - VDI and AES*

*Published by Digital Research*

This is the definitive document on GEM but contains many errors and is geared towards the 8086 version. Regrettably it is only available to registered developers.

## BASIC

The BASIC language is better served with books than the Atari hardware and software - there are a vast number of books that will try to teach you BASIC or act as a reference to some dialect and you should choose tirles that suit your requirements and tastes. Here's a selection that we like:

### *Microsoft QuickBASIC 2nd Edition*

*Douglas Hergert, Published by Microsoft Press*

ISBN 1-55615-125-X. Tutorial in nature and full of advice and suggestions on using the industry standard BASIC.

### *Microsoft QuickBASIC Handbook*

*Published by Microsoft Corporation*

The handbook that describes the Microsoft QuickBASIC compiler for the IBM. It contains details of the structured elements in the language.

### *Turbo Basic Owner's Handbook*

*Published by Borland International*

Accompanies the IBM version of Borland's Turbo Basic compiler. It contains explanations of the more structured elements in the language.



## 68000

### *16-Bit Microprocessors*

*Whitworth, Ian R. [1985]*

ISBN 0-00-383113-2, William Collins Sons & Co. Ltd., 8 Grafton Street, London W1X 3LA, U.K.

### *68000 Assembly Language Programming 2nd Edition*

*Kane, G., D.Hawkins and L.Leventhal [1987]*

ISBN 0-07-881232-1, Osborne/McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710, USA.

### *68000 Machine Code Programming*

*Barrow, David [1985]*

ISBN 0-00-383163-9, William Collins Sons & Co.Ltd., 8 Grafton Street, London W1X 3LA, U.K.

### *68000, 68010, 68020 Primer*

*Kelly-Bootle, Stan and Bob Fowler [1985]*

ISBN 067-22405-4, Howard W.Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

### *M68000 Family Programmer's Reference Manual*

*Motorola Inc. [1989]*

Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

### *Mastering The 68000 Microprocessor*

*Robinson, Phillip R. [1985]*

ISBN 0-8306-1886-4, Tab Books Inc., Blue Ridge Summit, PA 17214, USA.

*Microprocessor Systems: A 16-Bit Approach*
*Eccles, William J. [1985]*

ISBN 0-201-11985-4, Addison-Wesley Publishing Company, Reading, MA, USA.

*Programming the 68000*
*Williams, Steve [1985]*

ISBN 0-89588-133-0, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

*The MC68000 User's Manual 7th Edition*
*Motorola Inc. [1989]*

ISBN 0-13-567074-8, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

*The MC68020 User's Manual 2nd Edition*
*Motorola Inc. [1985]*

ISBN 0-13-566878-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

*The MC68030 User's Manual*
*Motorola Inc. [1987]*

Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

*The MC68881/MC68882 User's Manual*
*Motorola Inc. [1987]*

ISBN 0-13-566936-7, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.



## Algorithms & Data Structures

*Compilers: Principles, Techniques and Tools*
*Aho, Alfred V, Ravi Sethi and Jeffrey D. Ullman [1986]*

ISBN 0-201-10194-7, Addison-Wesley Publishing Company, Reading, MA, USA.

*Algorithms*
*Sedgewick, Robert [1988]*

ISBN 0-201-06673-4, Addison-Wesley Publishing Company, Reading, MA, USA.

*Data Structures and Algorithms*
*Aho, Alfred V, John E. Hopcroft and Jeffrey D. Ullman [1983]*

ISBN 0-201-00023-7, Addison-Wesley Publishing Company, Reading, MA, USA.

*Fundamental Algorithms*
*Knuth, Donald E. [1973]*

ISBN 0-201-03809-9, Addison-Wesley Publishing Company, Reading, MA, USA.

*Seminumerical Algorithms*
*Knuth, Donald E. [1981]*

ISBN 0-201-03822-9, Addison-Wesley Publishing Company, Reading, MA, USA.

*Sorting and Searching*
*Knuth, Donald E. [1973]*

ISBN 0-201-03803-X, Addison-Wesley Publishing Company, Reading, MA, USA.

# Appendix E
# New Features

Here is a list of most of the features of HiSoft BASIC 2 that are additional to our older BASICs, Power BASIC and HiSoft BASIC 1. Existing programmers may well find this list useful in making the most out of the package.

- New, multi-window editor with bookmarks, mouse block-marking, program launching and flexible user configuration options.
- Standard Compilation up to 50% faster; pre-tokenised files make development even faster.
- Complete resource construction set with conversion programs.
- Medium level debugger with the ability to view BASIC source code and use line numbers in debugging.
- Atari STE/DMA sound library including easy control of volume, tone etc.
- Easier desk accessory support - can even use `PRINT` now.
- Complete high level GEM Toolbox.
- `STATIC` arrays are now supported for faster array accesses.
- The running program can automatically warn you if you auto-dimension an array.
- Library functions may be called without `FN` giving much cleaner progams.
- TT library support for memory management and graphics.
- Compiler, Profiler and Debugger 'understand' about include files so that, for example, run-time error messages tell you in which file the error occurred.
- BIOS `RWABS` routine will use the AHDI 3 extensions if required.
- `OPEN` can now re-open the printer channel (256) so that you can re-direct `LPRINT` to disk.
- << and >> are unsigned shift operators which operate on both integers and longs (between addition and comparison in priority).

- IS is allowed in SELECT CASE for compatibility with MicroSoft BASIC.
- REDIM PRESERVE is a synonym for REDIM APPEND as per MicroSoft 7.1.
- RINSTR searches backwards in strings.
- Environment variable support for options and include path.
- STICK, STRIG and MOUSE functions have been extended to work with the STE analogue ports.
- Special TT version of the compiler that generates inline code for the maths co-processor (numerically intensive programs can run 100 times faster than with software routines) and also takes advantage of the new 68030 instructions.
- TOS utilities that do not clear the screen on startup can now be written.
- The integrated compiler can output code to TT RAM and the user has full control over the GEMDOS load bits.
- You can now link with Devpac 3 and Lattice C assembly language and C code and even access BASIC global variables directly from assembly language.
- The compiler can remove inter-sub program jumps giving more compact programs.
- Reserved words may be dis-abled make it easier to quickly port programs from other systems and to help check that code is portable.
- The return Stack can be much larger so that massively recursive programs can be run.
- Option to suppress compiler output apart from the
- CURDIR$ returns the current directory
- ENVIRON environment variable support
- BLOAD and BSAVE can now be used to channels.
- GETCOOKIE Atari Cookie Jar support.
- FORMATx$ for the flexibility of PRINT USING but to a string rather than directly to a channel.
- FREEFILE for writing file handling sub-programs that you can slot in anywhere.
- LTRIM$ and RTRIM$ for trimming spaces from strings.
- SPEEKx, SPOKEx for easy peeking and poking of supervisor memory.
- Window margins used by the PRINT statement are now user configurable via the SYSTAB table.



# Index

## D



## E



## F

## G



## H



## I



## J



## K



## L



## M



## N



## O

## P



## Q



## R



## S

## T



## U



## V



## W



## X

# Notes